The Acceleration of Structural Microarchitectural Simulation via Scheduling

David Aaron Penry

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

November 2006

Abstract

Microarchitects rely upon simulation to evaluate design alternatives, yet constructing an accurate simulator by hand is a difficult and time-consuming process because simulators are usually written in sequential languages while the system being modeled is concurrent. Structural modeling can mitigate this difficulty by allowing the microarchitect to specify the simulation model in a concurrent, structural form; a simulator compiler then generates a simulator from the model. However, the resulting simulators are generally slower than those produced by hand. The thesis of this dissertation is that simulation speed improvements can be obtained by careful scheduling of the work to be performed by the simulator onto single or multiple processors.

For scheduling onto single processors, this dissertation presents an evaluation of previously proposed scheduling mechanisms in the context of a structural microarchitectural simulation framework which uses a particular model of computation, the Heterogeneous Synchronous Reactive (HSR) model, and improvements to these mechanisms which make them more effective or more feasible for microarchitectural models. A static scheduling technique known as partitioned scheduling is shown to offer the most performance improvement: up to 2.08 speedup. This work furthermore proves that the the Discrete Event model of computation can be statically scheduled using partitioned scheduling when restricted in ways that are commonly assumed in microarchitectural simulation.

For scheduling onto multiple processors, this dissertation presents the first automatic parallelization of simulators using the HSR model of computation. It shows that effective parallelization requires techniques to avoid waiting due to locks and to improve cache locality. Two novel heuristics for lock mitigation and two for cache locality improvement are introduced and evaluated on three different parallel systems. The combination of lock mitigation and locality improvement is shown to allow superlinear speedup for some models: up to 7.56 for four processors.

Acknowledgements

First, I thank my advisor, David I. August, for his support and mentoring. As the founder and leader of the Liberty Research Group at Princeton University, he has provided an exciting environment in which I and others could explore better methods of microarchitectural modeling. Along the way he has consistently pushed towards more functionality and speed in the Liberty Simulation Environment; the automatic parallelization of LSE presented in this dissertation is a direct, successful result of this pressure.

The insights of various faculty members at Princeton and elsewhere have improved this work. Sharad Malik helped frame my understanding of the scheduling problem by asking about models of computation at the beginning of the development of LSE. Olivier Temam asked the important question of why I should bother with parallelization at all. Committee members Doug Clark, Kai Li, and J. P. Singh have made many useful observations on methodology. Vivek Pai stepped in to pinch-hit at a crucial moment.

It's been a pleasure working with the entire Liberty Research Group. Manish and Neil Vachharajani are responsible for most of the features which make the Liberty Simulation Environment so useful for microarchitectural modeling. Additional work was contributed by Jason Blome and Jonathan Cheng. Ram Rangan and Julia S. Chen contributed several valuable modules and models used in this dissertation. And various office mates have put up with my noisy work habits.

A very special thank you goes to Daniel Gracia Pérez, Giles Mouchard and Olivier Temam for giving me access to the source code of FastSysC and the models they used for its evaluation.

Thanks go to the American taxpayer who has contributed through a National Defense Science and Engineering Graduate Fellowship and later through various NSF project grants (CCR-0133712, CNS-0305617, IGERT-9979230) administered by Professor August. I also thank the Upton family for the Francis Upton Fellowship which supported me for four years.

I thank also my sister-in-law, April, for the tray-table and my father-in-law, Reed, for lending his office for a week before a paper deadline. And finally, I thank my wife, Sherise, for her support in this endeavor. Despite my crazy working hours, trips to the hospital, and ever-present deadlines, she's managed to hold everything together, doing far more than her fair share. It couldn't have been done without her. Dw i'n dy garu!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer microarchitects need to consider, evaluate, and select from many design alternatives when planning a new microprocessor. Evaluation through hardware prototyping is expensive and time-consuming, leading microarchitects to seek other techniques to evaluate design alternatives. The most commonly used technique is software prototyping, or simulation.

Because microarchitects must consider many design alternatives, the ease of creation and modification of microarchitectural simulators is very important. If it is too difficult to create or modify simulators, microarchitects will consider fewer alternatives and make poorer decisions. At the same time, many individual simulation runs must be made for each alternative, causing the speed of the simulators themselves to be very important. Faster simulators allow microarchitects to evaluate their design alternatives more thoroughly and to have higher confidence in their decisions.

This work focuses on automatically accelerating microarchitectural simulation while using simulation frameworks which allow the microarchitect to easily and rapidly specify, reuse, and modify simulators. Automatic acceleration is needed because such simulation frameworks may produce simulators which are slower than those created using other techniques. The combination of automatic acceleration techniques with simulation frameworks provides the microarchitect with simulators which are both fast and easy to modify and enables better design decisions. The thesis of this dissertation is that simulation speed improvements can be obtained by careful automatic scheduling of the work to be performed by the simulator onto single or multiple processors.

1.1 Microarchitectural Simulator Design

Microarchitects primarily evaluate hardware performance. The goal of this evaluation is to determine how long software will take to run on the proposed hardware. Because microprocessors are mostly designed as synchronous (clocked) logic,[1] performance is determined at the clock cycle level. Full within-clock-cycle timing information for each wire in the processor is not necessary. Neither is full fidelity to each wire's behavior in the hardware; abstractions such as modeling an adder as simply an addition operation on two integers rather than as a collection of bit-level full adder blocks are possible, and even desirable to improve clarity and simulation speed. Therefore, microarchitectural simulation is normally high-level cycle-based simulation; only the signal values at clock boundaries are modeled, and those signals may contain multiple physical wires.

The most common approach to creating microarchitectural simulators is to write a simulator in a sequential programming language such as C or C++[99]. While these languages are familiar to most microarchitects, they are not well-suited for writing simulators because there is an inherent semantic mismatch between the natures of microprocessors and sequential programs. Microprocessors are inherently concurrent and structural in nature but sequential programming languages are not. Even the use of object-oriented programming to model some of the structure fails to capture the essentials of the system: method calls have very different semantics from assigning values to signals. This mismatch between hardware and simulator is called the mapping problem. The mapping problem has been shown to lead to errors in understanding simulator code and increased development and modification times[103].

Vachharajani et al.[102] propose that microarchitects avoid the mapping problem by using concurrent-structural simulation frameworks (also known as structural simulation frameworks) to construct a concurrent, structural model of the microprocessor. This model reflects the concurrency and structure of the hardware and thus requires no mapping. The framework then generates a simulator from the model. Vachharajani et al. have shown[103] that users of such a framework are able to better understand and more rapidly modify models than sequential simulators. We have shown elsewhere[77] that a complex processor can be modeled with a high degree of accuracy in a matter of weeks using a structural simulation framework.

[1]Truly asynchronous microprocessors are relatively rare; their simulators may require detailed timing information. Such designs will not be considered further in this dissertation.

Structural simulation frameworks present a notion of concurrency to the user. The semantics governing this concurrency are known as the model of computation implemented by the framework. The model of computation has numerous implications for the ease with which systems can be modeled, the run-time overhead of manipulating signals, and the ability to schedule the concurrent execution to run efficiently.

1.2 The Problem of Speed

Simulation speed is of great importance to microarchitects; the faster the simulator, the more thoroughly design alternatives can be evaluated, leading to better decision making. Unfortunately, structural simulation may hurt simulator performance. A hand-coded simulator, while difficult to write, may be able to rely upon designer intuition, specialize code, have simple signal semantics through variables, or inline signals into registers. Reusable components in structural simulation may be over-generalized, having more functionality, state, or communication than is required for a particular model. Thus it is important to improve structural simulation speed. Speed is determined in large part by how well concurrent model behavior within a clock cycle is scheduled onto the processors running the simulation. Different means of scheduling have different overheads and performance.

Simulation speed will only grow in importance in the future. Simulation speed depends strongly upon the complexity of the design being simulated. This complexity has increased from generation to generation of microprocessors. Table 1.1 shows the total and estimated non-cache transistor counts in succeeding generations of Intel®microprocessors. (Cache array transistors are excluded because increased cache sizes do not increase microarchitectural complexity.) While transistor count is not a perfect metric of microarchitectural complexity, it shows the general trend well: complexity has increased greatly over the years. Fortunately, the host machines upon which microarchitects run simulation have become correspondingly faster, keeping pace with this growth in design complexity.

Recently chip manufacturers have been forced to halt or even reduce the growth in core complexity because of power dissipation issues. They have turned instead to replication of cores across the chip, creating chip multiprocessors. This replication of cores increases the overall

		Transistors	
Processor	Year	Total	Excluding cache[2]
4004	1971	2,300	2,300
8008	1972	3,500	3,500
8080	1974	6,000	6,000
8086	1978	29,000	29,000
Intel286	1982	134,000	134,000
Intel386™	1985	275,000	275,000
Intel486™	1989	1,200,000	800,000
Intel®Pentium®	1993	3,100,000	2,300,000
Intel®Pentium®II	1997	7,500,000	5,500,000
Intel®Pentium®III	1999	9,500,000	7,900,000
Intel®Pentium®4	2000	42,000,000	28,000,000
Intel®Pentium®4 (Northwood)	2002	55,000,000	28,000,000
Intel®Pentium®4 (Prescott)	2004	125,000,000	73,000,000

Table 1.1: Size of Intel®Processors

microarchitectural complexity or amount of work which a simulator must do, but the individual cores do not increase in performance as quickly as they have in the past. Indeed, some manufacturers (such as Sun with the N1 processor) are *decreasing* individual core performance as they replicate the cores. The result is that microarchitectural simulators will not be able to keep pace with the growth in design complexity unless they can utilize multiple cores or processors. In other words, microarchitectural simulators must become parallel programs.

While the simplest way to parallelize a microarchitectural simulator is to simply run individual experiments concurrently, doing so will not provide the best performance in several situations. One situation occurs when latency of individual simulation runs matters more than throughput of simulation runs. Another occurs when an increase in available cache size caused by using multiple processors improves performance. Finally, and very importantly for chip multiprocessors, when hardware resources are shared, individual experiments running concurrently may interfere negatively with each other in ways that a parallelized simulator could avoid. Thus it is important for the simulator itself to be parallelized and to use multiple processors.

Parallelization of a simulator written in a sequential language takes the already difficult and time-consuming task of writing a simulator and further complicates it. Fortunately, a structural simulation framework can exploit information about concurrency inherent in the structure to au-

[2]Source: Intel Corp./ PCStats.com / pcguide.com. Cache transistor count estimated as $6 * numberofbits$.

tomatically parallelize the simulator as it is generated. The acceleration which this parallelization provides to the simulator will depend upon how well the concurrent nature of the system is scheduled onto multiple processors.

1.3 Research Objectives and Contributions

Good scheduling of model concurrency in a structural simulator is very important to overall structural simulation speed, whether this scheduling is done for a single processor or multiple processors. This dissertation aims to find efficient and effective means to statically schedule model concurrency onto a single processor for the commonly used models of computation for structural microarchitectural simulation and to automatically parallelize the generated simulators and efficiently statically schedule model concurrency onto multiple processors.

The literature has addressed the scheduling of simulation model concurrency onto single processors mainly within the context of logic simulation. The first contribution of this dissertation is to evaluate these previous scheduling mechanisms within the context of structural microarchitectural simulation for a simulation framework which uses the Heterogeneous Synchronous Reactive (HSR) model of computation. The dissertation contributes improvements to these mechanisms to make them more effective or more feasible for microarchitectural models. These improvements include:

- Dependence information enhancement to improve schedule quality
- Dynamic subschedule embedding to improve scheduler time complexity
- A novel technique for reducing repeated work
- Forced invocations to permit dynamic scheduling techniques

This dissertation also contributes a proof that the most commonly used model of computation, the Discrete Event model of computation, can be efficiently statically scheduled when models are restricted in ways that are common in microarchitectural simulation. This result contradicts assumptions made by previous researchers[46, 36].

Other researchers have introduced the idea of parallelizing microarchitectural simulators[33, 67, 17, 6]. This previous work has involved manual parallelization of a simulator written in a sequential language. Other work has focused upon parallelization of simulators using the Discrete Event model of computation[11, 14, 45, 30, 27]. No previous work has addressed the parallelization of simulators which use the Heterogeneous Synchronous Reactive model of computation. Thus another contribution of this dissertation is the first method to automatically parallelize structural simulators which use this model of computation.

Finding efficient schedules for mapping concurrency onto multiple processors is equivalent to the problem of multiprocessor task scheduling with precedence constraints, resource constraints, sequence dependence, and communication costs. Additional contributions of this dissertation are four new heuristics for solving this problem:

- Two lock mitigation heuristics to reduce time spent waiting to obey resource constraints.
- Two clustering heuristics to improve cache locality and reduce the effects of sequence dependence.

1.4 Organization of the Dissertation

Chapter 2 provides background information about structural simulation frameworks and discusses models of computation commonly used in structural simulation systems. It provides a brief overview of the most common structural simulation systems as well as details of the Liberty Simulation Environment[102, 101], which is the structural simulation framework used in this dissertation.

Chapter 3 investigates methods of scheduling concurrent models for execution on a single processor. It evaluates four previously proposed techniques and proposes enhancements required to make these techniques feasible and effective for microarchitectural simulation. It discusses the conditions under which scheduling techniques designed for one model of computation can be applied to other models of computation. It also shows that additional information supplied at component library creation time can improve the scheduling results.

Chapter 4 shows that the generated simulators can be automatically parallelized. It further shows that efficient parallelization requires the use of scheduling techniques which can reduce the amount of time spent waiting for locks and improve cache locality. These are equivalent to solving the static multiprocessor task scheduling problem with precedence constraints, resource constraints, a form of sequence dependence, and communication costs. Four novel heuristics for doing so are presented and evaluated for three types of parallel computer systems.

Finally, Chapter 5 summarizes what this dissertation shows about scheduling to accelerate uniprocessor and parallel microarchitectural simulation and muses upon directions for future research.

Chapter 2

Structural Simulation

Writing a simulator for a hardware system is a modeling process; the simulator writer models the timing and functional behavior of the hardware. Hardware systems consist of components which operate concurrently and communicate with each other through signals. Yet simulators are generally written in a sequential programming language such as C or C++. Important hardware concepts such as concurrency and communication are not directly representable in these languages. Similarly, many of the important concepts of a sequential language such as global variables and method calls do not have good analogs in hardware[24]. This fundamental mismatch between the system being modeled and the technology used to model it has been identified as the **mapping problem**. Vachharajani, et al. have shown[103] that this problem is one of the primary reasons it is difficult to write an accurate simulator.

Structural simulation frameworks have been proposed as a means to solve the mapping problem and thus reduce the development effort needed to model a system at the microarchitectural level. This chapter explains the important characteristics of structural simulation frameworks. One of the most important of these concepts is the set of rules controlling the concurrent behavior of the system; these rules are known as the **model of computation**. A detailed description is given of the Liberty Simulation Environment, which is the framework used in this dissertation.

Framework	Type of System	Model of Computation	Intended Use
Verilog	HDL	Discrete Event	RTL
VHDL	HDL	Discrete Event	RTL
SystemC	C++ library	Discrete Event	RTL to system-level
FastSysC[36]	C++ library	Zero-delay Discrete Event	Microarchitecture
Expression[39]	ADL	Strict	Microarchitecture
HASE[19]	Framework	Discrete Event	Microarchitecture
Asim[24]	C++ library	Clocked Synchronous	Microarchitecture
LSE[102]	Framework	HSR	Microarchitecture

Table 2.1: Characteristics of common structural simulation frameworks

2.1 Structural Simulation Frameworks

A structural simulation framework is a modeling framework in which the behavior of a hardware system is modeled using concurrently executing components communicating primarily through signals.[1] Such a framework removes the mapping problem by removing the need to perform the mapping from hardware concepts to simulator code manually. The model corresponds directly to the hardware or the user's mental understanding of the hardware. The framework is responsible for performing the mapping and generating a simulator binary from the model.

There are several examples of structural simulation frameworks. Some are designed specifically to be frameworks. Others are architectural description languages (**ADLs**) or hardware description languages (**HDLs**) which can be used as frameworks. An overview of structural simulation frameworks is given in Table 2.1. For more detailed descriptions of these frameworks, see [99]. In this dissertation, the Liberty Simulation Environment (LSE) will be used for evaluation of scheduling techniques; it is described in more detail in Section 2.3. The listed frameworks will be used as examples in the following pages as the elements, organization, execution, and models of computation of structural simulation frameworks are described.

2.1.1 Structural Model Elements

This subsection presents the essential elements of a structural model. These elements are components, signals, time, and state. The terminology introduced is that used by the Liberty Simulation Environment but the concepts apply to all structural simulation frameworks and equivalent ter-

[1]This is a more expansive definition than that taken by [99] and includes some so-called pseudo-structural frameworks.

minology in other frameworks is given in footnotes when terms are defined. Concepts specific to the Liberty Simulation Environment are described later in Section 2.3.

Components

A structural model contains **components**[2] instantiated from a library of modules. A **module**[3] is simply a template or class for generating a component; a component is an instance of a module. Each module contains one or more **codeblocks**.[4] A codeblock is a block of sequential code which reads input signal values, produces output signal values, and/or updates state. Individual codeblocks logically execute concurrently and generally may make no assumptions about the execution of other codeblocks. The nature of this concurrency depends upon the model of computation and will be discussed in Section 2.2.

The purpose for having multiple codeblocks per module is to either organize the module code more conveniently or to share state between codeblocks in an instantiated module. For example, data signals and flow control signals may be handled by different codeblocks but still depend upon some state of the component. The granularity and composition of codeblocks is typically left up to the user. Codeblocks are commonly considered **black boxes**; the framework does not have the ability to analyze the internal logic of codeblocks, though it may provide some rules which constrain codeblock behavior.

Two primary goals of many structural simulation frameworks are reuse and flexibility. The intent of the structural framework is to make it possible to reuse modules or to modify them quickly and easily. Frameworks may have reuse-enhancing features such as default flow control behavior, parametric polymorphism (the ability to make a module which works with any data type), or algorithmic parameters. Modules in libraries are typically written with many parameters allowing them to be used in many situations.

---

[2]Components are known as instances in SystemC, Verilog, VHDL, and Asim.

[3]Modules are known as entities in HASE and VHDL and units in Expression.

[4]Codeblocks are called threads in HASE; methods in Asim and Expression; processes in SystemC and VHDL; and tasks, structured procedures, or continuous assignments in Verilog.

Signals

Components are connected together using **signals**.[5] These signals are generally typed and can take on a rich set of values – e.g. integers, arrays, or records – allowing for higher-level abstractions than physical wires. Connections are made between **ports** of components. Components communicate with each other primarily through the signals connected to their ports, though some frameworks may also allow shared state, which must be accessed carefully to prevent race conditions.

Signal values are produced by the execution of codeblocks; a codeblock assigns a value to the signal. Constant signal values are possible; for analysis purposes they can be considered to be assigned by a special codeblock within the framework. More than one codeblock may potentially assign to a signal, just as more than one codeblock may potentially read the value of a signal. Codeblocks which assign to a signal are said to be connected to codeblocks which read the signal's value. Occasionally the act of assigning to a signal is described as "sending a message" on the signal, though this terminology will not be used in this dissertation.

A very important issue is whether there are restrictions on the patterns with which codeblocks or instances can be connected in a system. If the signals are considered as edges of a directed graph and the codeblocks or instances as the vertices, are there any directed cycles in the graph? Some frameworks allow such cycles; others do not. If cycles are not permitted, some useful patterns of connection between components are not allowed, reducing reuse of modules. For example, a simple two-signal handshake between two components might not be possible because the data signal and an acknowledge signal would form a cycle in a graph of instances.

Time

Microarchitectural simulation requires time measurements at clock-cycle granularity; it does not require timing information within a clock cycle. Thus the only time values of interest are zero and integral numbers of cycles. Time can be introduced as either delays upon the signals or latency taken by codeblocks to produce signal values. In this dissertation, all signals have zero intrinsic

[5] Signals are a special case of channels in SystemC; they are called links in Hase, connections in Expression, and ports in Asim.

delay and codeblocks assign values to signals with some delay; this causes the new value of the signal to not "take effect" until after the delay has passed.

Codeblocks which never assign with zero delay will be called **E-codeblocks**. All other codeblocks assign with zero delay to at least one signal and are denoted **W-codeblocks**. In some frameworks, W-codeblocks may only assign with zero delay and only E-codeblocks may assign with non-zero delay. The E-codeblock and W-codeblock nomenclature stems from the portion of the main simulation cycle in which each codeblock type is normally invoked, as described in Section2.1.3; E-codeblocks are invoked in the **E**nd-of-cycle portion while W-codeblocks are invoked in the **W**ithin-cycle portion.

Note that different frameworks provide the user with different means to indicate that a codeblock is an E-codeblock or a W-codeblock. In LSE there is a direct syntactic distinction through a naming convention. In Expression and Asim the signals carry delay annotations which indicate the amount of delay codeblocks use when assigning to each signal. In Verilog, VHDL, HASE, and SystemC a codeblock which is sensitive to a clock (i.e. runs when a clock changes value) is an E-codeblock and all other codeblocks are W-codeblocks.[6]

A signal A is said to **computationally depend** or simply **depend** upon another signal B if A is assigned to with zero delay by some codeblock and that assignment is control-dependent or data-dependent upon a read of the value of signal B within the execution of the codeblock.

A **zero-delay cycle** is a cycle in a graph of W-codeblocks where edges represent signals which are assigned to with zero delay.

State

Components may contain state. This state is normally updated by E-codeblocks with non-zero delay so that state changes are not seen until the next clock cycle. It can be possible for a W-codeblock to successfully update state if either the construction of the codeblock or guarantees provided by the framework prevent that state change from affecting any signal values in the current clock cycle. Some frameworks provide these guarantees. If a W-codeblock updates state

[6]Codeblocks in Verilog, VHDL, and HASE are able to assign with arbitrary delays, but this capability is not generally used in cycle-granularity simulation.

inappropriately, a **race condition** occurs where the signal values depend upon the invocation order of the codeblocks.

Some frameworks (LSE, Verilog, SystemC, and possibly Asim or Expression) may allow codeblocks of different components to share state. This state sharing often occurs through libraries such as instruction set emulators. State sharing may also occur implicitly in all frameworks whenever I/O is performed by the simulator. Race conditions may occur very easily when state is shared between codeblocks of different components and the user must reason carefully about how and when state is updated.

2.1.2 Organization of a Structural Model

Structural models are usually organized in a hierarchical fashion. The framework allows the user to define modules as a collection of interconnected sub-modules. Instantiating modules creates an instance hierarchy of components. Components which are at the lowest level of the hierarchy are known as **leaf components**.[7] Frameworks generally "flatten" this hierarchy to generate a netlist of inter-connected leaf components.

This model organization reflects the nature of most microprocessor designs. A microprocessor can be thought of as a very large state machine that is too large to reason about as a single machine. Instead, the microprocessor is partitioned into multiple peer state machines which communicate with each other. This partitioning is reflected in the codeblocks of the components. E-codeblocks correspond to **Moore** state machines; their outputs are dependent only upon their state[51]. The stored signal values for the next cycle are state for the Moore state machine, as is user-defined internal state which the E-codeblock updates. W-codeblocks can be thought of as **Mealy** state machines when they reference or update state because their outputs depend upon both inputs and state. W-codeblocks which reference and update no state are combinational logic.

In a synchronous processor design, all state machines update their state simultaneously according to clock signals. The key characteristic of a synchronous design is that no state exists whose update is not controlled by a clock. Many structural simulation frameworks are intended to simulate only synchronous designs. However, it is possible for state uncontrolled by clocks

[7] Leaf components are called atomic components in HASE.

to **emerge** from the interconnections of the combination logic of the system and cause a design to become non-synchronous. This may happen even though individual codeblocks are combinational; individually combinational logic blocks might not be joinly combinational after they are connected. The interconnection patterns which a framework can support can be limited by emerging state. Thus the remainder of this subsection explores this phenomenon more fully.

State emerges when under some combination of input values and assumptions about delay, a signal depends upon itself. Malik[65] characterizes the situations under which a cyclic logic circuit remains combinational: for a combinational circuit, primary outputs always reach stable values for all combinations of primary input values and those output values are completely determined by the values of the primary inputs for any combination of gate and wire delays. This is equivalent to saying that neither the eventual stability nor the stable value of any primary output is affected by the value of an emergent state signal. I extend Malik's characterization to non-boolean signal values: a circuit is combinational if the stability of and stable value of no primary output is affected by the value of an emergent state signal.

When state arises from the interconnections, the model is no longer synchronous, as that state is not controlled by the clock. This gives rise to an operational definition of whether a model is synchronous: if all old explicit state values are treated as primary inputs and all new state values are treated as primary outputs, then if and only if the combined combinational portions of all state machines in the model are jointly combinational, the model is synchronous.[8]

It is possible in microarchitectural modeling for designs to appear to not be synchronous even though there exists a synchronous logic-level implementation. This is because the types used in microarchitectural modeling are larger than a single bit and every bit of a signal is considered to be involved when computing dependences. Such a situation is shown in Figure 2.1. Here signal Z depends upon itself through the codeblock when the signal is considered as a whole. However, a logic-level implementation would not appear to depend upon itself because no bit of Z actually depends upon itself. In this dissertation, those models which admit a logic-level synchronous implementation are called **logically synchronous** models and the subset of the logically synchronous models which still appear synchronous at the microarchitectural signal level are termed

[8]Unreachable states should not be considered when checking whether the model is jointly combinational. If unreachable states are considered, then some models will be designated as not synchronous which always show synchronous behavior in practice.

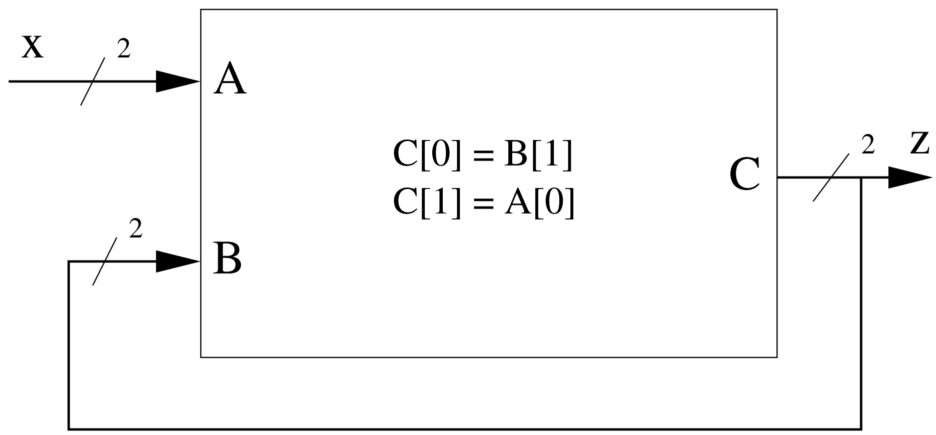


Figure 2.1: A logically synchronous but microarchitecturally asynchronous model

microarchitecturally synchronous. This distinction will be important later when the ability of different models of computation to connect codeblocks will be considered.

2.1.3 Structural Model Execution

The framework is responsible for producing a simulator to execute the model. To do so, the framework must map the concurrent behavior of the model into a sequential simulator. The structure of the main simulation loop for most frameworks can be seen in Figure 2.2. The frameworks which use the Discrete Event model of computation have a slightly different main loop in which the within-cycle and end-of-cycle steps are merged into one step. However, as execution actually alternates between W-codeblocks and E-codeblocks, the structure given in Figure 2.2 still applies.

Before simulation begins, there is an **INIT** step which initializes data simulation structures and state. Then, during each clock cycle of simulation, the following steps are taken:

within-cycle During the within-cycle step, W-codeblocks are invoked to compute signal values for the current cycle and potentially for following cycles. In some frameworks, codeblock invocations may not be repeated. In others, invocations may be repeated until signal values converge to a stable state. In yet others, invocations may be repeated, but signal values are not guaranteed to converge, nor is this step guaranteed to terminate.

end-of-cycle During the end-of-cycle step of simulation, E-codeblocks are invoked to update state and compute signal values for following next cycle.

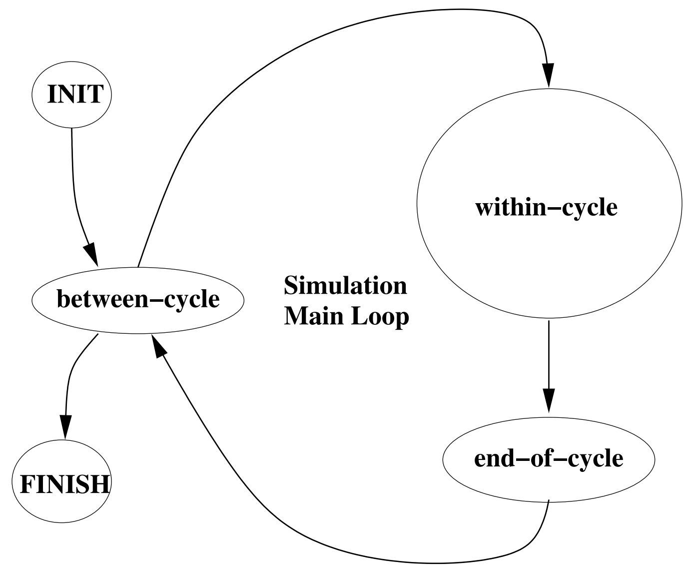


Figure 2.2: The simulation loop

between-cycle During the between-cycle step of simulation, the framework performs bookkeeping such as incrementing time and applying signal values produced with some latency as the initial signal values for the next clock cycle.

The within-cycle step and the end-of-cycle step are often combined, though it is clearer to think about their execution as separated. After simulation ends, there is a **FINISH** step, which typically reports summary statistics.

This simulation loop has been described for a hardware model in which there is a single clock. When there are multiple clocks, the same loop is maintained, but different iterations correspond to different clocks. For example, if there are two clocks A and B with the same period but some amount of skew such that B lags A, then the odd iterations of the main loop will reference clock A and the even iterations clock B. Only end-of-cycle codeblocks corresponding to the clock being referenced by a particular iteration are invoked in that iteration. This dissertation deals only with single-clock models, but the techniques are easily in this way to multiple-clock models.

The order in which codeblocks are executed and the manner in which signal values evolve over the course of execution must obey certain rules which define the semantics of the system. These rules are known as the **model of computation**. To avoid confusion with the model of hardware which the user creates, models of computation will often be referred to as **MoCs**.

2.2 Models of Computation (MoCs)

Researchers do not agree upon the definition of the term "model of computation." One defines model of computation as "the set of processes and process networks that can be constructed by a given set of process constructors and combinators."[44, p. 108] Yet another defines it as "the model of time employed ..., the supported methods of communication between concurrent processes, and the rules for process activation."[38, p. 41]. In this dissertation, model of computation is defined as:

Definition 1. *A **model of computation (MoC)** is the set of rules which control:*

- *The values signals may assume.*
- *The latencies codeblocks may use when setting signal values.*
- *The order in which codeblocks execute.*
- *The order in which signals are evaluated.*

The MoC has five important implications for ease of modeling and simulation performance of a structural simulation framework:

1. A MoC may restrict the granularity of components or the way in which components can be interconnected, thus limiting reuse or flexibility.

2. A MoC may allow a W-codeblock to update state. This may both simplify and accelerate the code. An example of a module which can benefit from this ability is a superscalar register renamer, where the rename of later instructions depends upon the rename of earlier instructions.

3. A MoC affects the ease with which users can share state between codeblocks by permitting or restricting this sharing.

4. A MoC affects the overhead of manipulating signals; some MoCs require more work to be done when a signal value is assigned or read than others.

MoC	Limitations	Within-cycle State Update	Within-cycle State Sharing	Overhead
Clocked Synchronous	granularity and connectivity	N/A	no	buffering
Strict	connectivity	yes	if ordered	limited buffering
Discrete Event	none	no	no	limited buffering, comparison
Heterogeneous Synchronous Reactive	monotonicity	yes	if ordered	limited buffering, $\perp$ checks

Table 2.2: Characteristics of MoCs used in structural simulation frameworks

5. A MoC determines whether static or dynamic scheduling of codeblock execution is possible and desirable. The subject of Chapter 3 is efficient scheduling methods for the different MoCs and a more detailed explanation of possible scheduling methods for each MoC is deferred until then.

While there are many MoCs, there are several particular MoCs that are commonly used in structural microarchitectural simulation. Table 2.2 summarizes these MoCs. They are described below in greater detail.

2.2.1 The Clocked Synchronous MoC

The simplest model of computation is the Clocked Synchronous MoC[93]. It is used by the Asim[24] simulation framework. This MoC enforces a clock cycle of delay along any communication path between codeblocks; equivalently, all codeblocks must assign with non-zero delay. This leads to the following set of rules:

1. All signal assignments have non-zero delay.
2. Signals may take on only one value per clock cycle.
3. Only E-codeblocks may exist (a corollary of rule 1).
4. All codeblocks must execute exactly once per cycle.

These rules imply that codeblocks may only model Moore finite state machines. Such a restriction greatly reduces modeling flexibility; it is not possible to compose blocks of combinational logic. Some frameworks using this model of computation attempt to relax this restriction by allowing the user to create a within-cycle step of simulation by calling methods of the components. Doing so requires that the user manually write code to make these calls in a correct order; changes to the model structure result in a need to modify the calling code.

Codeblock writers may take advantage of the single execution per cycle guarantee to simplify some codeblocks and may update state in any codeblock immediately. Sharing of state between codeblocks is not feasible because there is no execution ordering between the codeblocks.

All signal assignments must be buffered by the framework to comply with rule 1, but this may not be a large overhead as the granularity of modeling is typically not extremely fine with this model of computation.

2.2.2 The Strict MoC

The Expression[39] simulation framework uses a model of computation which this dissertation terms the Strict MoC. This name was suggested by an observation that Expression obeys the **strictness** property of function languages[79] – all inputs to a function must be evaluated before the function is called. The Strict MoC allows zero-delay assignments, but requires that there be no zero-delay cycles between W-codeblocks.

It has the following rules:

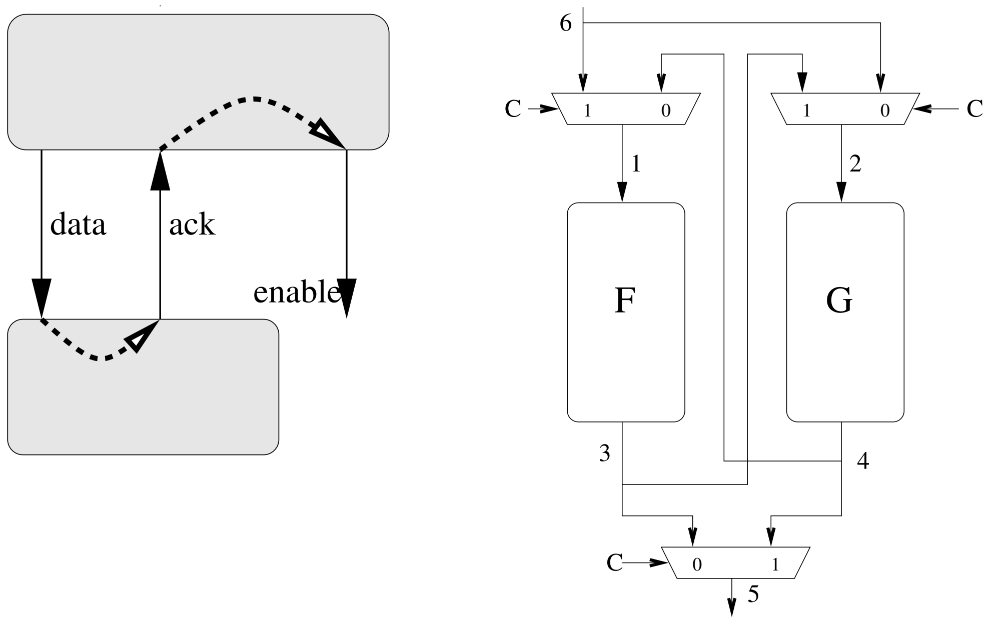
1. Signal assignments may have delay of either zero or an integral number of cycles.

2. Signals may take on only one value per clock cycle.

3. No codeblock may execute until all its inputs have taken on their final value for the clock cycle. This rule is the strictness property. This rule prevents zero-delay cycles among the W-codeblocks; it also implies that W-codeblocks need be executed no more than once; Expression guarantees that W-codeblocks will execute exactly once.

4. E-codeblocks must execute exactly once per cycle.

These rules allow individual codeblocks to model any finite state machine or combinational logic, but they limit the patterns in which components can be connected. In particular, zero-delay cycles between W-codeblocks are not possible. While this restriction may seem unobjectionable



(a) Spurious zero-delay cycle (b) Unrealized zero-delay cycle [65, Fig. 2]

Figure 2.3: Seemingly zero-delay cycles

at first glance, it is too restrictive because seemingly zero-delay cycles appear more frequently in microarchitectural modeling than might be expected. This can happen for two reasons:

1. A W-codeblock may calculate multiple output signals using different input signals. A seeming zero-delay cycle involving such a codeblock may not in fact involve a true cycle in computational dependences. As an example, consider Figure 2.3(a). Here the upper component places a data request to the lower component and receives an acknowledge signal in return. It then creates an enable signal based upon that acknowledge signal. It may be very convenient to write the upper module with one codeblock to handle all this behavior. However, there appears to be a zero-delay cycle between the codeblocks which is not actually present in the true computational dependences.

2. A zero-delay cycle may exist statically, but not dynamically. One situation in which this may occur when is large datapath elements are interconnected with routing elements. For example, Figure 2.3(b) shows a situation where a control signal selects between function $F \circ G$ or $G \circ F$. The zero-delay cycle is never dynamically realized. A similar situation may occur in distributed arbitration logic.

In either case the hardware cannot be described without changing the codeblocks, reducing flexibility and reuse. For example, in Figure 2.3(a), the upper codeblock would need to be split into two codeblocks: one which generates the data signal and one which handles the ack-to-enable calculation. In Figure 2.3(b), the hardware cannot be described at all in the strict MoC without duplicating at least one of F or G.

Codeblock writers may take advantage of the single execution per cycle guarantee to simplify some codeblocks and may update state in any codeblock immediately. Sharing of state between E-codeblocks is difficult because there is no ordering guarantee between the codeblocks. Sharing of state between W-codeblocks is possible between two codeblocks whose execution order is guaranteed because there has been dataflow between them; sharing between W-codeblocks and E-codeblocks is also possible.

Signal assignments which have non-zero delays must be buffered by the framework to comply with rule 1. Other signal assignments can be simple assignments to variables.

2.2.3 The Discrete Event MoC

The Discrete Event MoC is used by simulation frameworks such as Verilog, VHDL, SystemC, and HASE[19] and is the least restrictive model of those presented here. Instead of restricting delays to an integral number of clock cycles, delays may be any amount of time. The rules are:

1. The assignment of values to signals produced in a codeblock is delayed by a user-assigned amount of time. Some frameworks may permit this time to be zero; others may cause it to always be at least a **delta** amount which is reported as zero elapsed time.

2. If delta-delays are used, signals may take on only one value per evaluation step, but there may be multiple such steps per timestep. Otherwise, a signal may take on multiple values per timestep.

3. Codeblocks must execute in the evaluation step in which any input signal to which they are **sensitive** changes value. Codeblocks may not execute multiple times due to a single change in a sensitive input.

The Discrete Event MoC does not distinguish between W-codeblocks and E-codeblocks. Instead, an E-codeblock can be recognized by the fact that it is sensitive to an explicit clock signal. As a practical matter, while the simulation main loop need not distinguish between W-codeblocks and E-codeblocks, execution usually alternates between E-codeblock activity and W-codeblock activity. The FastSysC[36] implementation of SystemC relies upon this behavior to improve simulation speed for cycle-accurate models.

These rules allow individual codeblocks to model any finite state machine or combinational logic. Connections patterns are not constrained in any way; the examples given in Figure 2.3 present no difficulties. In fact, the overall model need not be synchronous, nor need it even be guaranteed to converge to a stable value in any timestep. However, because microarchitectural modeling is usually synchronous, it is common to restrict the Discrete Event MoC in this case to allow only zero and integer amounts of delay; when this is done, explicit clock signals are also removed and E-codeblocks are treated differently from W-codeblocks. With the restricted model of computation, some asynchronous models cannot be modeled without rewriting of codeblocks, but all microarchitecturally synchronous and logically synchronous models can be.

As there are no guarantees that a W-codeblock will not be invoked many times per clock cycle, nor any guarantee that a particular invocation will produce the final value of a signal for the clock cycle, W-codeblocks may not update state. E-codeblocks may update state. State may not be shared between W-codeblocks or between E-codeblocks as no execution order within each class is guaranteed. However, W-codeblocks may read state written by E-codeblocks.

The Discrete Event MoC typically has a high degree of overhead. All signal assignments with non-zero delays, including delta-delays, must be buffered by the framework. Only zero-delay assignments may be direct assignments to variables. The final rule implies that new signal values must be compared to old signal values to determine whether the receiving codeblock must be scheduled for invocation unless a static scheduling scheme is used; these comparisons may be quite expensive to perform when the signal values are complex data structures.

2.2.4 The Heterogeneous Synchronous Reactive (HSR) MoC

The Heterogeneous Synchronous Reactive MoC was proposed by Edwards[21] as a model appropriate for use in component-based embedded software systems design. His motivation was to allow arbitrary patterns of connectivity while guaranteeing that signal values converge to stable values in each timestep after a predictable maximum number of codeblock invocations. This MoC is used by the Liberty Simulation Environment.

The original HSR MoC model was derived from the Synchronous Reactive (SR) MoC. **Synchronous** here means that only zero-delays are allowed. **Reactive** means that the system only executes due to a change in inputs. **Heterogeneous** means that codeblocks can be considered as block boxes. For microarchitectural simulation, the HSR model is extended to include non-zero delays on assignments; this can be viewed as the non-zero-delay assignments providing input stimuli to a proper HSR system formed from the W-codeblocks. The resulting rules are:

1. Signal assignments may have delay of either zero or an integral number of cycles.

2. Signal values are members of a pointed complete partially-ordered set (a **poset**). What this means is that there exists a partial order for the signal values where there is a unique $\perp$ value which is less than all the other values. In frameworks which use this MoC, the partial order is very simple: the $\perp$ value is less than all other values and all other values are not comparable. Only this particular partial order is considered in the discussion below and in this dissertation. With this partial order, the $\perp$ value is interpreted as "not yet computed".

3. W-codeblocks must be **monotonic**. In other words, if $\vec{x}$ and $\vec{x}'$ are two vectors of input signal values, the codeblock transfer function $\vec{f}(\vec{x})$ (a functional description of the zero-delay signal assignments a W-codeblock makes) must satisfy the property: if $\vec{x} \leq \vec{x}'$, then $\vec{f}(\vec{x}) \leq \vec{f}(\vec{x}')$. Note that the ordering relation for a vector of signal values is the natural extension of the ordering relation for a single signal value: only if all components of one vector are less than or equal to the corresponding components of another vector is the first vector less than or equal to the second vector.

   This monotonicity constraint, while complex-looking, is actually quite easy to implement with the simple partial order assumed. The constraint can be satisfied if whenever an input

A	B	Non-Strict	Strict
True	True	True	True
True	False	False	False
True	$\perp$	$\perp$	$\perp$
False	True	False	False
False	False	False	False
False	$\perp$	False	$\perp$
$\perp$	True	$\perp$	$\perp$
$\perp$	False	False	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Table 2.3: Comparison of strict and non-strict AND-gates

whose value is truly necessary to calculate an output has the $\perp$ value, the output remains $\perp$. In other words, if a necessary input has not yet been computed, then the output should not be computed.

4. The final value of a signal produced by a W-codeblock is that given by the least fixed point solution of the system of codeblock transfer functions.

5. E-codeblocks must execute exactly once per cycle.

The HSR MoC with the simple poset described allows individual codeblocks to model any finite state machine or combinational logic. Connection patterns are not constrained and seeming zero-delay cycles evaluate to the expected values; however if there is a true zero-delay cycle, the least fixed point solution contains $\perp$ values for the signals involved in the cycle. This is because if a signal truly transitively depends upon itself, $\perp$ is a fixed point for the signal value and no value can be less than $\perp$. Signal values which are $\perp$ in the solution are undesirable as they give no information about the actual value of the signal.

To avoid impeding reuse, outputs should be set to a non-$\perp$ value whenever this can be done without violating monotonicity. Not doing so stifles reuse because it causes more zero-delay cycles to be true cycles. For example, if all inputs must be non-$\perp$ before any output is assigned a non-$\perp$ value, the codeblock is actually "strict" and suffers from all the limitations of that MoC. To see how this might occur, observe Table 2.3, which gives two truth-tables for AND-gates. A non-strict AND-gate has the truth table given in the third column, but a careless module writer might be tempted to use the truth table given in the fourth column, yielding a strict AND-gate.

If a model is not microarchitecturally synchronous, there is some combination of state and input which resulted in state emerging from the connections. State can emerge only when there is a true zero-delay cycle, therefore, if a model is not microarchitecturally synchronous, some signal values will remain $\perp$. As a result, the HSR MoC with this poset cannot always compute meaningful signal values for a microarchitecturally asynchronous model without modifying the codeblocks.

On the other hand, if a model is microarchitecturally synchronous, there are no zero-delay cycles which affect outputs or new state. In such a model, due to monotonicity, $\perp$ signals will only occur if there is a zero-delay cycle which affects no outputs or new state, some input signals are $\perp$ or a codeblock always produces an $\perp$ output value and is thus useless. Note that state with $\perp$ values would cause $\perp$ signals, but that would imply that the previous cycle computed an $\perp$ value for the state, which would inductively imply one of the other conditions or an $\perp$ initial value for the state.

Thus if a signal value remains $\perp$, one of the following situations has occurred:

- The model is microarchitecturally asynchronous.
- The model is microarchitecturally synchronous but has a zero-delay cycle with no effect on output and state.
- The model is microarchitecturally synchronous but has $\perp$ input signals.
- The model is microarchitecturally synchronous but contains a useless codeblock.
- The model is microarchitecturally synchronous but has $\perp$ initial state.

All of these conditions should be rare; certainly the last three could be considered bugs. Thus the HSR MoC will compute meaningful (non-$\perp$) values for each signal for most microarchitecturally synchronous models. Those for which it does not are likely to contain errors.[9] The examples given in Figure 2.3 are microarchitecturally synchronous and do not present problems to frameworks using the HSR MoC. Note, however, that the HSR MoC with the simple partial

[9] Frameworks can use the presence of $\perp$ values as a debugging aid to indicate places where the model may be in error.

order used here will *not* compute meaningful values for some signals for logically synchronous but microarchitecturally asynchronous models.

Codeblock writers may simplify some W-codeblocks and improve performance by taking advantage of the fact that once a signal is no longer $\perp$ it will not change again. A simple check of an output signal value can thus skip the re-execution of code needed to compute that value. Immediate state update is also possible if output signals which depend upon that state have already been given values which are not $\perp$. Sharing of state between W-codeblocks is possible between two codeblocks whose order of assigning non-$\perp$ values to signals is guaranteed because there has been dataflow between them.

Signal assignments which have non-zero delays must be buffered by the framework to comply with rule 1. Other signal assignments can be simple assignments to variables, however the modeled values must be extended to include the $\perp$ value, which in most implementations results in the variable consisting of two objects: a bit field indicating $\perp$ and the desired value. The HSR MoC also requires some additional overhead when using signal values as this use is often guarded by a check for the $\perp$ value.

2.3 The Liberty Simulation Environment (LSE)

The Liberty Simulation Environment (LSE)[102] is used throughout this dissertation to evaluate scheduling techniques. LSE is a structural simulation framework designed to promote reuse and flexibility in microarchitectural modeling. This section describes a user's view of LSE and characteristics which influence the scheduling techniques introduced in the next two chapters.

2.3.1 A User's View of LSE

To the user, LSE is a tool chain with a library of core modules. The user instantiates components, parameterizes and customizes them, and connects them together with typed signals. LSE takes as its input a model specification and module templates and produces an executable simulator. This flow is shown in Figure 2.4.

Model specifications are written in a language called the Liberty Structural Specification Language (LSS)[100]. This language provides constructs for instantiating components, setting

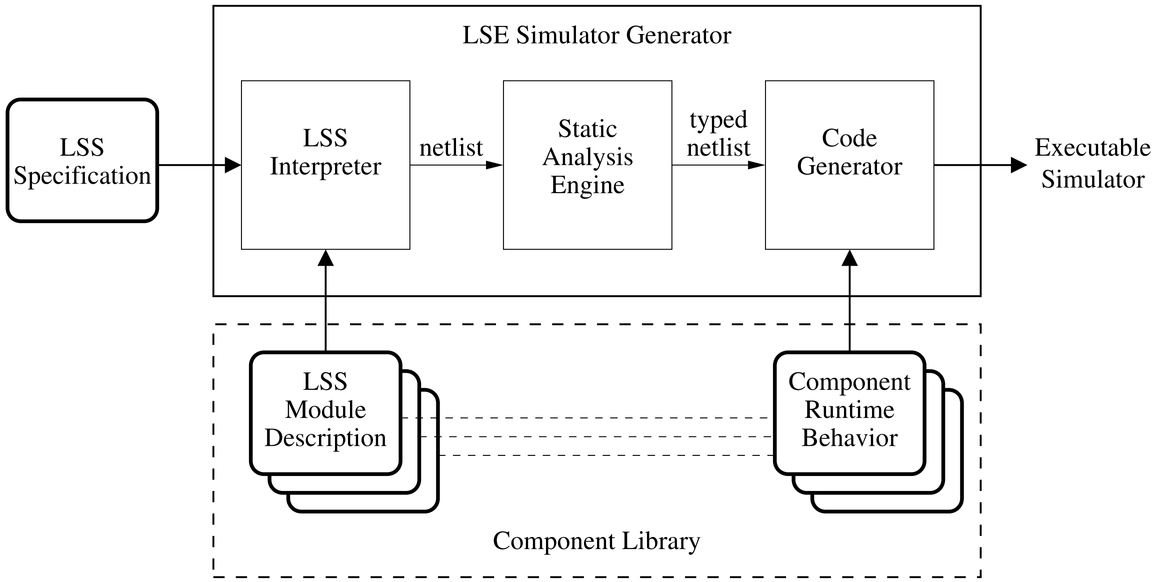


Figure 2.4: Overview of LSE [99, Fig. 5.1]

parameters, making connections between ports, and managing hierarchy. This language is interpreted to produce a netlist. Unlike most netlisting languages, this language has very rich semantics, supporting complex data types, type inference, functions which construct structure, and model error checking.

Modules have an interface description written in LSS, but their behavior is specified in a behavioral specification language (BSL), which is at present a stylized version of C++ with added API calls. The BSL code can be thought of as a template for the instantiated components (though C++ templates are not the implementation mechanism). The code generator combines the netlist with the BSL to generate C++ code for each component, the framework APIs visible to the user, and the schedule of codeblock invocations. This C++ code is highly specialized for the model.

The model of computation used by LSE is the HSR MoC. While this would seem to be mainly of concern to writers of modules, many library modules allow algorithmic customization and allow the user to manipulate signal values directly as part of this customization. As a result, all users of LSE must understand the monotonicity requirement of the HSR MoC.

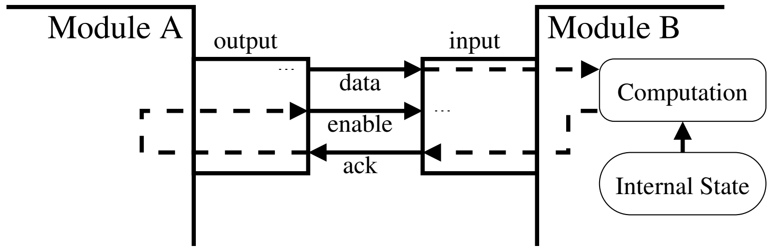


Figure 2.5: Default flow control signals in LSE[102, Fig. 4]

2.3.2 Writing Modules

Modules either come from the standard module library or are written by users. To write a module, a user writes an interface description for it in LSS and a behavioral description in BSL. The behavioral description is a C++ class with stereotypical names for the methods which are to be considered as codeblocks. The internal logic of codeblocks is not analyzed by LSE and thus codeblocks are black boxes. A module may have no more than one E-codeblock, but may have many W-codeblocks. There are three types of W-codeblocks.

The first type is called a **handler**. Handlers are used to "handle" a change in input signal values on a specific port. If multiple connections are made to a port, there the port is said to have multiple **port instances**. When a handler is invoked, it is passed a parameter indicating the port instance whose signal changed value. The single handler for a port in the module code is considered to generate as many codeblocks as there are port instances for that port. A handler must assign values to all output signals which are computationally dependent upon its input signals and whose values can be determined at the time of the handler's invocation.

The second type of W-codeblock is called a **phase function**. There is at most one phase function per module. This function is considered to be the handler for all signals which do not have handlers. A phase function must assign values to all output signals whose values can be determined.

The third type of W-codeblock is called a **control function**. Every connection in LSE is point-to-point (there is no fanout) and actually consists of three signals. These can be seen in Figure 2.5. The first signal carries data. The other two signals are boolean flow control signals. Modules from the standard library implement a standard handshaking protocol which makes it easy to connect together modules and obtain reasonable flow-control behavior. Control functions are W-codeblocks which can modify the flow control signals flowing into and out of a port. Control functions are specified by the user in LSS while customizing a component. Much like handlers, a control function is passed the port instance and thus the control function's code generates as many codeblocks as there are port instances.

Only E-codeblocks may assign values with non-zero delay to signals; thus W-codeblocks may only assign with zero delay. E-codeblocks are split into two portions. The first portion only updates state, but cannot assign any signal values. The second portion assigns signal values, but it does not run until the within-cycle step of the next clock cycle. These second portions are called **phase start functions** and are treated as W-codeblocks with no input signals for scheduling purposes.

Chapter 3

Scheduling for Uniprocessor Structural Simulation

A major factor determining the speed of a structural simulator is the quality of scheduling of concurrent codeblock execution onto a single processor. This chapter investigates efficient codeblock scheduling techniques for the models of computation used by structural microarchitectural simulation frameworks. It begins with a discussion of how scheduling affects simulator performance. Previously proposed scheduling techniques for each of the models of computation and how they can be applied to other models of computation are described. It then introduces the challenges which microarchitectural simulation presents to scheduling techniques, and proposes novel enhancements to address these challenges. The scheduling techniques with the proposed enhancements are evaluated in the context of a structural microarchitectural framework which uses the Heterogeneous Synchronous Reactive (HSR) MoC.

3.1 The Relationship of Scheduling to Performance

Structural simulation frameworks generate simulator code. This code consists of a set of codeblocks which the user has specified as well as logic which schedules and invokes these codeblocks for execution. The codeblocks are considered to execute concurrently by the user; thus the scheduling and invocation logic must map this logically concurrent execution onto invocations of the codeblocks on a single processor running the simulator.

Codeblocks are divided into two classes: those which assign values with some delay to all signals which they produce (E-codeblocks) and those that assign values to some signals with zero-delay (W-codeblocks). Scheduling the execution of E-codeblocks is quite simple; any schedule which invokes each E-codeblock exactly once is correct. However scheduling the execution of W-codeblocks is more challenging, and may require multiple invocations of the same codeblock.

There are two basic approaches to scheduling W-codeblocks: dynamic and static. Dynamic scheduling is the simpler of the two. A mutable **invocation list** of codeblocks to invoke is maintained. When a signal changes value during execution, the codeblocks which receive that signal are added to the list. Codeblocks are removed from the list when they are invoked. When the list is empty, the within-cycle execution is finished. After within-cycle execution has completed, end-of-cycle execution executes all the E-codeblocks in any order.

There is an additional form of dynamic scheduling, called **two-pass scheduling**. (The form described above is known as **one-pass scheduling**[98].) In two-pass scheduling, there are two lists: an invocation list and a signal change list. Assignments of values to signals do not directly modify the signal values or the invocation list; instead, assignments add elements to the signal change list. The scheduler has two passes: during the signal update pass it applies signal changes from the signal change list and adds elements to the invocation list. During the invocation pass it invokes codeblocks, which add elements to the signal change list. One-pass scheduling is generally more efficient as it does not require buffering of signal changes, and is thus preferred in most situations, but it does not support the delta-delay variety of the Discrete Event MoC.

Static scheduling is more complex and requires that the framework compute a schedule which obeys the rules of the MoC and which will guarantee that all signals obtain their correct values by the end of the within-cycle phase of execution. This schedule may be computed during generation of the simulator code or it may be computed at simulator initialization time. The resulting schedule is an immutable invocation list of codeblocks to be invoked; within-cycle execution is a simple traversal of this list, invoking the indicated codeblocks.

Scheduling affects performance through two main factors: the overhead of manipulating the list of codeblocks to be invoked and the number of codeblock invocations performed. In general, static scheduling has lower overhead than dynamic scheduling because the list operations are both

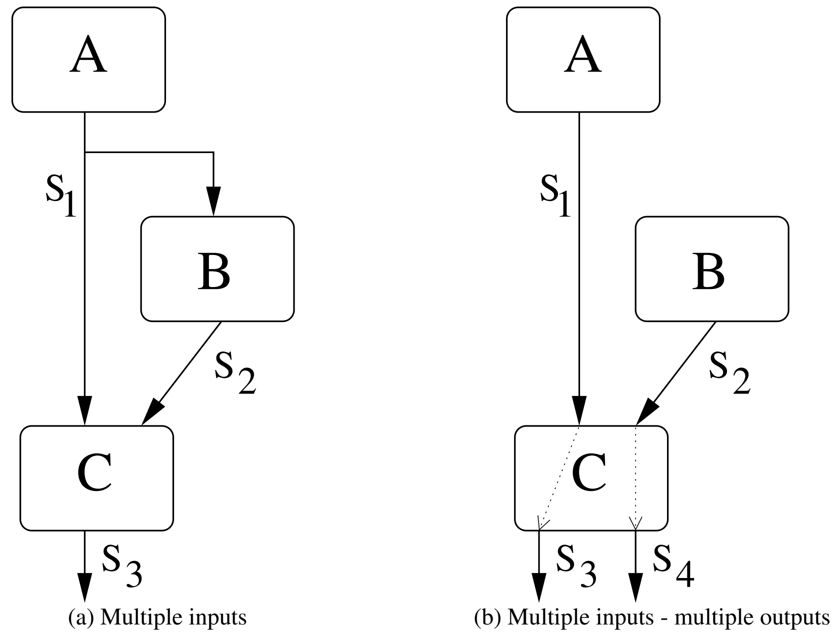


Figure 3.1: Invocation reduction opportunities for static scheduling

fewer and simpler. The number of codeblock invocations is less simple to analyze because there are several competing factors.

At first glance, dynamic scheduling should have fewer invocations than static scheduling because dynamic scheduling is driven directly by changes to signal values; a codeblock need only be dynamically scheduled when an input signal changes, in contrast to static scheduling which requires that all codeblocks be evaluated on all cycles. However, this analysis is misleading in that it depends upon the activity factor, the MoC, and implementation details. The activity factor is a measure of how often signals change values; if they change values frequently, dynamic scheduling does not have many fewer invocations. Also, the Heterogeneous Synchronous Reactive MoC requires that signal values be "reset" to an $\perp$ value at the beginning of each cycle, making it impossible to make this comparison across cycle boundaries. The result is that for this MoC, each codeblock receiving a signal runs at least once per cycle. Furthermore, it is possible while using static scheduling to guard invocations of codeblocks such that they are only invoked if signals changed. This results in an increase in overhead, but may reduce the number of codeblock invocations significantly.

Static scheduling may also result in fewer codeblock invocations than dynamic scheduling when certain interconnection patterns between module instances exist. Figure 3.1(a) shows an example of one of these situations. The vertices in this directed graph are codeblocks and the edges are signals which are assigned with zero delay by the source codeblock and read by the target codeblock. In a dynamic scheduling approach, the execution of codeblock A produces a value for signal S_1, which adds blocks B and C to the invocation list. Suppose that block C is then invoked before block B. It will compute some value for signal S_3 based upon the current value of signal S_2. When block B executes afterward, it may change the value of signal S_2, which then will cause block C to be invoked again to recompute the value of signal S_3. Static scheduling is able to determine that the schedule ABC would prevent this re-invocation. The potential for this behavior is present whenever a W-codeblock has multiple inputs which are generated by other W-codeblocks; the codeblock may be invoked before all of its inputs are ready and thus must be invoked again later.

Static scheduling may also reduce codeblock invocations when codeblocks have multiple output signals which do not depend upon precisely the same input signals. Figure 3.1(b) shows an example of this situation. It is very similar to the previous example, except that codeblock C has two outputs; the dotted lines show the true computational dependence within this codeblock. In this situation, dynamic scheduling will add blocks C to the invocation list when the value for signal S_1 is computed by the execution of block A. If block C is invoked before block B, then block C will need to be invoked again after B computes signal S_2, but if C is invoked after B, only one invocation of C is required to compute both output signals. Static scheduling may be able to detect this condition and **coalesce** the two invocations of C required to calculate the two independent outputs to produce the schedule ABC.

Of course, scheduling is not the only factor affecting simulator performance. Modeling style, and particularly the choice of component granularity, may greatly influence speed. As granulary increases, performance increases due to decreasing inter-component communication overhead, but reuse and flexibility also decrease. The user is ultimately responsible for choosing the level of granularity. However, improving the scheduling as proposed in this dissertation should lessen the severity of the speed-to-reuse tradeoff, thus lessening barriers to reuse.

Speed may also be affected by good software engineering practice and optimization within the framework, as was demonstrated dramatically by the FastSysC project[36]. The scheduling techniques discussed in this chapter are all evaluated within the context of a single structural framework so that differences in implementation efficiency between frameworks do not affect comparisons between scheduling techniques.

3.2 Scheduling and Graph Terminology

The scheduling techniques used in this dissertation and in the literature operate upon directed graph representations of the codeblocks and signals in the hardware model in order to produce schedules. In general, these graphs are multigraphs – they may have multiple edges connecting the same vertices – but they are typically referred to simply as graphs. There are three such directed graphs derived from the hardware model:

- The **connection graph** contains vertices which represent the module instances and edges which represent connections between the module instances. Each edge corresponds to some signal; each signal is represented by as many edges as the fanout of the signal.
- The **codeblock graph** contains vertices which represent the W-codeblocks. There are edges between two codeblocks for each signal which the source codeblock may generate with zero-delay and the target codeblock may read.
- The **extended codeblock graph** contains vertices which represent all W-codeblocks and E-codeblocks. There are edges between two codeblocks for each signal which the source codeblock may generate with zero delay and the target codeblock may read.
- The **signal graph** contains vertices which represent each signal in the model. There is an edge between two vertices if the target signal computationally depends upon the value of the source signal.

A **path** in a directed graph is a sequence of edges such that the source vertex of an edge is the same as the target vertex of the previous edge. A path can also be denoted as a sequence of vertices such that there is always an edge from each vertex to the next vertex. A **simple path** is a path in which no vertex is repeated.

The notion of how "far" a vertex in a graph is from a source or sink vertex is used by several of the scheduling techniques. While different terminology is used by different authors, this dissertation defines the following concepts of distance:

tlevel The *tlevel* of a vertex is the maximum length (in number of edges) of all paths from source vertices to that vertex. Source vertices have a *tlevel* equal to zero.

blevel The *blevel* of a vertex is the maximum length (in number of edges) of all paths from that vertex to sink vertices Sink vertices have a *blevel* equal to zero.

A **cycle** is a path of length greater than zero which begins and ends at the same vertex. Note that when cycles are present, *tlevel* and *blevel* cannot be defined for all vertices of a graph.

The asymptotic runtime complexity of various algorithms will be specified using O-notation, Θ-notation, or Ω-notation where V stands for the number of vertices and E stands for the number of edges in the graph. As there are several graphs defined for a system, subscripts will be used to denote the graph which is being described. These subscripts are C for codeblock, E for extended codeblock, and S for signal. For example, E_S is the number of edges in the signal graph and V_C is the number of vertices in the codeblock graph.

Some techniques use the **strongly-connected components (SCCs)** of a graph. A strongly-connected component of a graph is a maximal subgraph such that every vertex is reachable from every other vertex. SCCs can be found quite easily in $\Theta(V + E)$ time by an algorithm due to Tarjan[95] which consists of two depth-first searches. If each SCC of a graph is shrunk to a single vertex, the resulting **component graph** is acyclic. A topological sort of the component graph produces a **topological order** of the SCCs. Tarjan's algorithm has the additional property of returning the SCCs of a graph in topological order.

A **back edge** of a graph is defined with respect to some depth-first search of a graph as an edge whose source vertex is finished before its target vertex during the depth-first search. Back edges with respect to a strongly-connected component decomposition of a graph are those found by the first depth-first search in Tarjan's algorithm. Tarjan's algorithm extended to report back edges will be referred to as FIND-SCCS-AND-BACK-EDGES and continues to require running time of $\Theta(V + E)$.

3.3 Related Work

Because scheduling is so important to the performance of structural simulators, the topic has been widely discussed in the literature. In this section, the most relevant related work is presented. The section is organized by model of computation.

3.3.1 Scheduling the Clocked Synchronous MoC

The clocked synchronous model of computation does not permit W-codeblocks, thus the within-cycle schedule is quite uninteresting; it is empty. The end-of-cycle schedule is arbitrary as long as each E-codeblock is invoked once.

In Asim[24], which uses this MoC, it appears that end-of-cycle scheduling is done at initialization time, though details are unclear.

While this MoC seems rather trivial, it is important because the other MoCs considered have E-codeblocks which have these same scheduling constraints. Also, there are opportunities for optimization even of the end-of-cycle schedule.

One opportunity for optimization of the end-of-cycle schedule has been previously introduced in a very different context. The idea is to schedule the codeblocks so as to increase instruction and/or data cache locality. This possibility was was introduced by Philbin et al.[80], who dealt with cache locality for independent tasks on a uniprocessor. This possibility is not explored further in this dissertation.

Another seeming opportunity is to use guards to prevent invocation of E-codeblocks whose input signals have not changed during the cycle. This turns out to not be feasible, as nearly all frameworks built on any MoC allow E-codeblocks to have internal state which should be updated on each clock cycle, requiring invocation of the codeblocks. A framework could allow the user to specify that state changes due to an end-of-cycle codeblock's execution will not affect output signals for some number of cycles and provide a means to "batch" these changes into one invocation; this would allow guarded execution. The Liberty Simulation Environment requires that the

E-codeblocks explicitly inform the framework of the earliest cycle when state changes may cause the output signals to be different. This information could be used for guarded invocation; instead, it is used it to determine whether any clock cycles can be skipped entirely. If no codeblock reports that output signals could be different in the future, the simulation terminates.

3.3.2 Scheduling the Strict MoC

The strict model of computation requires that each codeblock be executed after all of its inputs have reached their final value for the clock cycle. Because the codeblock multigraph is acyclic, any topological sort of the codeblock graph can be used to form a static schedule for the W-codeblocks. The E-codeblocks may be executed in any order and thus may be scheduled exactly like the Clocked Synchronous MoC.

Static scheduling for this MoC originated in the logic simulation community where it is known as **levelized logic simulation**. Logic simulation simulates the behavior of logic gates and is used extensively for many different VLSI design and test tasks. The term "levelized" comes from a particular form of topological sort which is used for scheduling. Gate evaluations are scheduled in order of their **level**. The netlist of gates is considered as a codeblock graph. Then the gate evaluations are scheduled either in increasing order of *tlevel* or decreasing order of *blevel* of the corresponding vertices in the graph. Levelization takes $\Theta(V_C + E_C)$ time.

For levelized logic simulation to work, the netlist of gates must be acyclic and the delay on signals and through the gates must be zero. As a result, though the logic simulation literature refers to this as zero-delay (or occasionally unit-delay) Discrete Event scheduling, it is really using the Strict MoC. These conditions are very commonly met in logic simulation; many of the uses of logic simulation do not require non-zero delays and designers try to avoid cycles (even if they are combinational) in the netlist of gates.

Levelized logic simulation was introduced by Wang et al.[104] in 1987 in a system known as SSIM using *tlevel* as the level, but similar levelization techniques were used simultaneously for fault simulation by Barzilai et al.[8]. These latter authors later used levelization with switch-level simulation[7]. Levelization was used several years earlier by Denneau to compile for hardware gate-level simulation[20]. Maurer and Wang extended the techniques to unit delays[66].

Guarded invocation is also possible when scheduling for the Strict MoC. Doing so requires that flags indicating that invocation is needed be maintained and that new values for signals be compared with the previous value to set these flags. Wang et al.[104] used this technique in SSIM; they call guarded invocation **selective trace**. Barzilai et al. called the technique **non-oblivious scheduling** in [8]. Guarded invocation is considered by many to be especially appropriate for logic simulation as the activity factor of logic simulation is very low; Wong et al.[109] measured the activity factor for a number of benchmark circuits and found it to be 1.2% on average.

At a higher level of abstraction than logic gates, Pétrot et al.[78] consider a set of communicating finite state machines (FSMs) with zero delay for communication between the FSMs. They show that a static schedule evaluating each FSM no more than once per cycle is possible if there are no combinational cycles in a graph in which the vertices represent the FSMs and edges represent Mealy signals (signals which are computationally dependent upon input signals). They then show that the static schedule can be found using a topological sort of the graph of FSMs. This graph is an extended codeblock graph where each FSM is a codeblock, and such a system meets the conditions of the Strict MoC. Doing so combines the within-cycle and end-of-cycle schedules. A novel feature of their work when compared to logic simulation is that they treat FSMs as black boxes (they do not need to know the whole behavior of the FSMs) with annotations which indicate which outputs are assigned to with zero delay. Also, they support multiple outputs per FSM. All FSMs may update state upon invocation.

Of the structural microarchitectural modeling frameworks, EXPRESSION[39] uses the Strict MoC. While details are not provided, EXPRESSION appears to use static scheduling performed at simulator initialization time using the extended codeblock graph and combined within-cycle and end-of-cycle schedules. Because EXPRESSION's static scheduling guarantees that W-codeblocks will be invoked at most once per clock cycle, W-codeblocks may update state.

3.3.3 Scheduling the Discrete Event MoC

The Discrete Event MoC is very widely used and has been studied extensively. The basic Discrete Event scheduling algorithm is dynamic. Distinctions between within-cycle and end-of-cycle codeblocks are not made in this style of scheduling; an end-of-cycle codeblock is simply a codeblock which is sensitive to a clock signal.

In its most general form, the Discrete Event MoC cannot be statically scheduled as the system is not guaranteed to converge to a stable state in all timesteps. Thus all attempts to statically schedule for this MoC must either restrict the MoC further to guarantee convergence or must combine dynamic and static scheduling techniques.

The earliest efforts to statically schedule the DE MoC restricted the codeblock graphs to acyclic graphs and all delays to zero, thus restricting the MoC to a Strict MoC. These have been dealt with in the previous subsection.

Hommais and Pétrot extend[40] their previous work with FSMs[78] to allow zero-delay cycles, thus extending the MoC considered to a zero-delay DE model and making their technique relevant to microarchitectural simulation. They generate a schedule by finding strongly-connected components (SCCs) in the extended codeblock graph and scheduling the evaluation of each SCC in topological order. Evaluation of a single-vertex SCC is simply the evaluation of the codeblock represented by that vertex. Evaluation of multi-vertex SCCs is performed by **Hamiltonian subschedule embedding**; a subschedule for the SCC is inserted into the overall schedule. This subschedule repeatedly invokes the codeblocks within the SCC until their output signals no longer change; this procedure is known as **relaxation**. The ordering for this subschedule is found by randomly selecting a vertex in the SCC which has an in-edge from outside the SCC. A Hamiltonian path (a path which visits each vertex exactly once) is then found starting at that edge. Not all SCCs may have Hamiltonian paths; if one cannot be found edges are added. They use an iterative algorithm presented in [18] to add the minimum number of edges necessary to cause the SCC to have a Hamiltonian path. Determining whether a Hamiltonian path exists was shown by Karp[49] to be an NP-complete problem.

Convergence for an SCC is checked only on the signals upon which signals outside of that SCC depend. No mention is made of the fact that the signals might never converge to a stable value in a poorly formed model. FSMs must be divided into W-codeblocks and E-codeblocks; the W-codeblocks are *not* allowed to update state.

French et al.[29] statically schedule event-driven simulation described in the Verilog language with a technique they call **static simulation**. This technique requires that all delays be constants known at compile time, but can handle complex and changing codeblock sensitivity lists. Their key observation is that a Discrete Event simulation is itself a non-deterministic finite state machine. The state is made up of the set of codeblocks (which they term **events** which are currently ready to execute; the set of codeblocks which could be triggered to execute due to dataflow or control flow; the current state of all variables; and codeblock executions already scheduled for the future. They generate code which explicitly echoes the form of this state machine. The state machine may have an enormous number of states; to keep them under control they limit memory values considered as unique states to provably constant values and use partial evaluation to find a fixed point and predicate the calling of codeblocks with the exact conditions under which the codeblocks should run. Levelization is found to be useful in deciding which state transition to take. They implement static simulation for a significant subset of the Verilog language in a compiler called VeriSUIF and demonstrate performance improvement relative to a commercial compiled DE system. Their technique does not permit black box modules because all state must be visible to the compiler.

Two additional techniques are particularly interesting because they are hybrid techniques which combine static scheduling with dynamic scheduling. These techniques are levelized event-driven scheduling and acyclic scheduling.

Levelized event-driven scheduling

Wang and Maurer[106] describe the LECSIM system, a zero-delay logic simulator which attempts to combine the reduction of invocations which static scheduling brings in situations like that of Figure 3.1(a) with the reduction in invocations of guarded execution and the ability to handle cycles in the codeblock graph. It does so by using a special data structure for the list of codeblocks to be invoked while doing one-pass dynamic scheduling. This data structure is shown in Figure 3.2. The data structure consists of an array of circular lists. Each of these lists corresponds to a *tlevel* in the codeblock graph. The data structure is traversed in increasing *tlevel* order. As codeblocks are inserted into the structure when their input signal values change, they

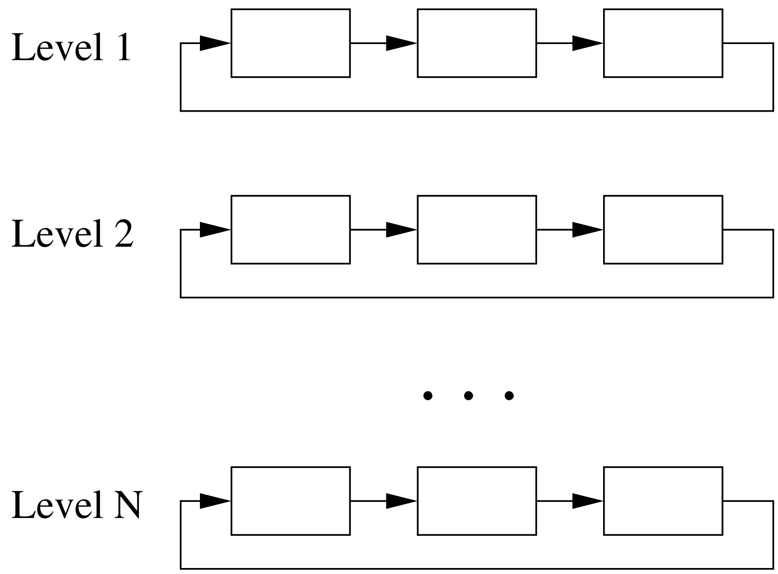


Figure 3.2: The LECSIM invocation list

are inserted into the list at the level corresponding to their *tlevel*. If this list is "earlier" than the current traversal position in the data structure, a flag is set indicating that another iteration of data structure traversal will be required. Their work did not provide for E-codeblocks, but E-codeblocks can be invoked once each in any order after the data structure traversal is finished.

The presence of cycles in the codeblock graph causes two problems: *tlevel* is not defined, and small cycles cause additional traversals of the entire data structure. To overcome this, LECSIM combines the breaking of edges as it calculates *tlevel* with subschedule embedding. Figure 3.3 shows the algorithm. First, the strongly-connected components of the codeblock graph are calculated. Large SCCs (those with more than some number of vertices) are hierarchically decomposed by removing some (but not all) back edges. Which edges are removed is not explained. Once there are no large SCCs remaining, new codeblocks are created for each small SCC with more than one vertex and these SCCs are replaced with vertices for the new codeblocks. This algorithm requires $\Omega(V_C + E_C)$ time; an upper bound cannot be definitely stated because hierarchical decomposition is not sufficiently well explained, though it seems likely that it would be $O(E_C(V_C + E_C))$.

The new codeblocks represent embedded subschedules for their corresponding SCCs in the original codeblock graph. They invoke their list of codeblocks repeatedly until no signal changes value or some limit on the number of iterations is reached. The limit is intended to detect situa-

LE-CALCULATE-TLEVEL(G, $smallSize$)
▷ G is the codeblock graph
▷ $smallSize$ is how big a "small" SCC can be

$SCCs \leftarrow$ FIND-SCCS-AND-BACK-EDGES(G)

▷ Break apart large components
while $\exists c \in SCCs\ s.t. |c| > smallSize$
  do Remove some back edges contained within c from G
    ▷ The following can be done incrementally by just considering c
    $SCCs \leftarrow$ FIND-SCCS-AND-BACK-EDGES(G)

▷ Replace small components with single vertices
foreach c **in** $SCCs$
  do if $|c| > 1$
    then
      Collapse c in G to a single vertex

Calculate *tlevel* for each vertex of G

Figure 3.3: Levelized event-driven scheduling

tions where the simulation never converges to a stable value within a clock cycle. The order of invocation within this subschedule is based upon the *tlevel* of each codeblock calculated with all back edges within the component removed.

Note that when a signal value changes which is read by a codeblock in an embedded subschedule that the subschedule must be scheduled for execution instead of the original reading codeblock. This can be accomplished by making a subschedule a reader of the union of signals read by its constituent codeblocks.

The codeblocks considered in LECSIM are either single gates or fanout-free blocks. A fanout-free block is a set of gates which feed each other where each gate's output signal has only a single reader. Within a fanout-free block, each gate is evaluated once in levelized order. The purpose of using fanout-free blocks is to reduce scheduling overhead, but they may lead to more gate evaluations because guarded execution is not used within them.

Levelized event-driven scheduling is shown to perform better than normal dynamic scheduling because it reduces codeblock invocations dramatically. When the activity factor is low, it outperforms levelized scheduling, again due to the reduction in codeblock invocations, but when the activity factor is high, the overhead of maintaining the data structure causes levelized event-driven scheduling to under-perform.

Gai et al. introduced a similar data structure in [31] where the objective was to suppress delta spikes (multiple changes in a signal value in the same timestep). However, they used a delta-delay version of the Discrete Event MoC, and thus required two-pass scheduling. They used the new data structure only for the invocation list and did not allow cycles in the codeblock graph.

Acyclic scheduling

Acyclic scheduling for a DE MoC was introduced by Gracia Pérez et al. for microarchitectural simulation in the context of the SystemC design language[36]. The objective is to obtain the reduction of invocations which static scheduling brings in situations like that of Figure 3.1(a) while allowing cycles in both the codeblock graph and the signal graph. The model of computation is restricted to delays of zero and one clock; W-codeblocks may only use zero delays. An implementation of a subset of SystemC which uses acyclic scheduling has been publicly released as FastSysC.

The basic idea of acyclic scheduling is to create a static schedule under the assumption that the signal graph is acyclic. Codeblocks are invoked according to this static schedule in each cycle, but this phase of invocation is followed by a dynamically scheduled one which should only find itself with work to do when the graph was not acyclic. Their rationale is that cycles in the signal graph should be rare in microarchitectural simulation and the use of modeler-supplied annotations can remove what cycles remain. Thus the dynamic scheduling merely provides a fall-back mechanism. This fall-back mechanism is also robust because incorrect annotations merely result in performance loss, not incorrect execution.

Unlike the techniques presented so far, acyclic scheduling uses the signal graph instead of the codeblock graph. This allows better static schedules, as cycles in the codeblock graph are not necessarily cycles in the signal graph, as seen in Figure 2.3(a). It means, however, that some form of invocation coalescing should be included in the scheduling to reduce unnecessary invocations of codeblocks which can produce multiple outputs, as seen in Figure 3.1(b).

The static scheduling algorithm is quite simple and is given in Figure 3.4. The signal graph is first made acyclic by removing edges. It is then topologically sorted, with the order formed using the *tlevel* of each vertex as in levelization. The sorted order is transformed into a list of codeblocks to invoke, but codeblock invocations are coalesced so that a codeblock is called no more than once for each signal it generates at the same *tlevel*. This algorithm can be made to run in $\Theta(V_S + E_S)$ time though the implementation used in [36] requires $O(V_S(V_S + E_S))$ worst-case time.

```
ACYCLIC-SCHEDULE(G)
  ▷ G is the signal graph
  if G is cyclic
     then Remove edges from G so that it becomes acyclic

  Calculate tlevel for each vertex of G
  SL ← vertices in G, ordered by tlevel

  CL ← EMPTY
  foreach codeblock c
       do last[c] ← −1

  foreach vertex n in SL
       do c ← n.codeblock
          if n.tlevel ≠ last[c]
            then Append c to CL
                 last[c] ← n.tlevel

  Return CL
```

Figure 3.4: Acyclic scheduling

Gracia Pérez et al. evaluate acyclic scheduling using simulators built from two models. The first model, a simple pipelined processor, shows a reduction of 22.7% in codeblock invocations and 1.39 speedup. The second model, an out-of-order superscalar processor, shows a reduction of 29.9% in codeblock invocations and 1.23 speedup.

An important issue is how to deal with the mixture of static and dynamic scheduling. When a signal value is changed, the system does not know whether to schedule the receiving codeblocks or not. In pure dynamic scheduling, it always should, but if the static schedule is sufficient, it should not. The solution used (but not described in detail) is to perform two-pass scheduling.

Changes to signal values are recorded but receiving codeblocks are not immediately scheduled. During static schedule execution, between each *tlevel* the signal values are updated and flags are set indicating that receiving codeblocks should be invoked. At the end of the static schedule, a transitional phase follows in which any codeblocks whose flags remain set are invoked. At the end of this phase, signals are updated and receiving codeblocks are added to an invocation list. Standard two-pass dynamic scheduling then follows, with alternating codeblock invocation and signal update passes.

They also use guarded invocation within the static portion of the schedule. This does not increase signal assignment overhead as it normally would because they already needed to compare signal values and set flags to support the dynamic phase of execution. They note that using a tight table-driven loop for scheduling leads to the test being a highly unpredictable branch; including the test actually removes most of the performance benefit from having skipped the invocations unless the scheduler code is not table-driven and is instead generated as a sequence of function calls with guards around them.

They do not indicate how back edges should be removed from a cyclic signal graph to make the graph acyclic. Indeed, both models used for evaluation were acyclic once annotations were supplied by the modeler and thus removal of these edges was not required. Examination of their code reveals that they perform a depth-first search of the signal graph, starting at an arbitrary vertex, removing any edges to previously visited vertices. This procedure may remove more edges than is necessary, as it does not solely remove back edges.

3.3.4 Scheduling the HSR MoC

The only work on static scheduling for the Heterogeneous Synchronous Reactive MoC is that of Edwards[21]. Because the final value of the signals produced by within-cycle codeblocks is given by the least fixed point solution of the system of codeblock equations, a simple scheduling scheme is to simply evaluate all within-cycle codeblocks repeatedly for a sufficient number of iterations to guarantee convergence. The sufficient number of iterations is the sum of the height of the posets of signal values across all signals. Such a scheme is quite inefficient.

Better results can be obtained through more careful scheduling. Edwards uses **partitioned scheduling** of signal evaluations; he partitions the signal graph into two subgraphs which are called the **head** and the **tail**. The partition and the choice of which subgraph is the head and which is the tail can be done arbitrarily, though the choices made will affect the efficiency of the schedule. A theorem originally due to Bekić[9] shows that the least fixed point solution of the system of codeblock equations can be found by repeatedly finding the least fixed point of the tail followed by an evaluation of the head until the signals in the head reach a fixed point and then finishing by finding the least fixed point of the tail once more. Thus a schedule can be produced through a divide-and-conquer algorithm which recursively finds a schedule for the tail and then iteratively repeats this schedule, interleaving evaluations of the head.

Edwards observes that when the head does not depend upon the tail, then the least fixed point can be found by finding the least fixed point of the head and then finding the least fixed point of the tail. He calls such a partition a **separable** partition. He shows that separable partitions are optimal in terms of the total number of signal evaluations required. Furthermore, a separable partition can always be found if any vertex in the graph is not strongly connected to all other vertices. Thus a strongly-connected component decomposition of the graph can be used to efficiently find the maximum number of separable partitions. Note that if there are no cycles in the signal graph, partitioning using the SCC decomposition results in a schedule which is a topological sort of the signal graph.

The resulting scheduling algorithm restricted to the case where no signal value's poset has height of more than one (which has always been the case for frameworks using this MoC) is given in Figure 3.5. The SCCs of the signal graph are found and scheduled in topological order. Components with more than one signal are partitioned in a fashion to be described later and the tail schedule is interleaved with evaluations of the head according to Bekić's theorem. Note that the evaluation of signals in the head can occur in arbitrary order; Edwards calls this parallel evaluation. The algorithm executes in $\Omega(V_S + E_S)$ time; the nature of the how partitions are chosen needs to be specified in order to give an upper bound to execution time.

The choice of partition for a strongly-connected component affects the number of signal evaluations; ideally, the optimal partition should be chosen. Unfortunately, there are $2^m - 2$ possible

```
PARTITIONED-SCHEDULE(G)
  ▷ G is the signal graph

  ▷ Sched is a schedule of signal evaluation
  Sched ← EMPTY

  SCCs ← FIND-SCCS(G)
  foreach C in SCCs in topological order
     do if |C| = 1
          then Append vertex in C to Sched
          else ▷ Do partitioning
               Choose H s.t. ∅ ⊂ H ⊂ C
               T ← C − H
               TailSched ← PARTITIONED-SCHEDULE(T)
               Append TailSched to Sched
               for i ← 1 to |H|
                   do Append a parallel evaluation of each signal in H to Sched
                      Append TailSched to Sched
  Return Sched
```

Figure 3.5: Partitioned scheduling

partitions to consider, where m is the size of the component. Edwards proposes a branch-and-bound search of the possible partitions both to reduce the number of hierarchical steps of partitioning and to prune partitions which produce worse behavior than simple iterative evaluation. Furthermore, he shows that if a non-separable partition is to be optimal, its tail must be separable and proposes a heuristic for forming partitions which will create only separable tails but may not find all such tails or the optimal partition. Even with this heuristic, the worst case running time is still $\Theta(V_S!)$. This is not just a theoretical worst case time; his experimental results show that in practice run-time depends exponentially on the number of signals, though it grows much less rapidly than the factorial.

The schedule of signal evaluations must be mapped into a schedule of codeblock invocations. This can be done by replacing each signal in the schedule with the codeblock which generates that signal. Yet a single codeblock invocation may compute multiple output signals, so an invocation per signal evaluation may be wasteful. Edwards suggests that invocations be coalesced using an algorithm which tries to move each signal evaluation earlier in the schedule until it can be coalesced with an earlier evaluation which invokes the same codeblock. He gives rules for moving signals which are very specific to his representation of schedules; a simpler equivalent algorithm is given in Figure 3.6. Note that Edwards' description is ambiguous about the order in which signals should be considered in the outer do-loop, but the example given in [21] considers the signals in order of their names. The running time for this coalescing is $O(VE)$ where V and E are measured on the signal graph.

SIGNAL-COALESCING(S)
  $\triangleright$ S is the original schedule: a list of invocations records which have three fields:
      $codeblock$ indicating the codeblock to invoke
      $signalList$ indicating the set of signals being produced by the invocation.
      $depList$ indicating $\bigcup_{v \in signalList}$ signals upon which v depends
    Signal lists initially contain a single signal.

  foreach signal sig
    do Find $s \in S$ s.t. $sig \in s.signalList$
      $pushableRange \leftarrow$ position of s
      foreach invocation t **in** S s.t. t is before s,
              moving backwards through S
        do if $\exists v \in t.signalList$ s.t. $v \in s.depList$
            then Exit inner **do** loop
          elseif $s.codeblock = t.codeblock$
            then $pushableRange \leftarrow$ position of t

      if $pushableRange \neq$ position of s
        then $\triangleright$ Coalesce s with earlier element
          $t \leftarrow$element of S at position $pushableRange$
          $t.signalList \leftarrow t.signalList \bigcup s.signalList$
          $t.depList \leftarrow t.depList \bigcup s.depList$
          Remove s from S

Figure 3.6: Signal-based invocation coalescing

Edwards does not used guarded invocation of codeblocks; all codeblocks are invoked as per the schedule. However, it would not be difficult to add guarded invocation at the cost of additional overhead to evaluate the guard before invoking the codeblocks and additional cost to set and clear flags.

Technique	Strict	General DE	Zero-delay DE	HSR
Levelization[104]	Origin	No	No	No
Dynamic scheduling	No	Origin	Yes	Modified
VeriSUIF[29]	Maybe	Origin	Yes	Yes
Hamiltonian subschedules[40]	Yes	No	Origin	Yes
Levelized event-driven[106]	Yes	No	Origin	Modified
Acyclic scheduling[36]	Modified	No	Origin	Modified
Partitioned scheduling[21]	Modified	No	Restricted	Origin

Table 3.1: Applicability of scheduling techniques to models of computation

3.4 Applying Scheduling Techniques Across MoCs

The previous section has presented a number of W-codeblock scheduling techniques for different models of computation. Some of these techniques may be useful for improving performance in MoCs for which they were not originally designed. Yet applying a scheduling technique outside of its "native" MoC is hazardous: it could create a schedule which does not result in the simulator computing the correct signal values. This section investigates which techniques may be safely applied to which MoCs. A summary is given in Table 3.1. Because so many techniques for Discrete Event were proposed with a zero-delay restriction, zero-delay DE is given its own column. The notation "origin" indicates that a technique was originally proposed for that model of computation. The Clocked Synchronous Model of Computation is not considered further because it does not have W-codeblocks.

Levelization is an important technique and is incorporated into other static and hybrid scheduling techniques, but by itself requires acyclic codeblock graphs, and thus can be applied directly to only the Strict MoC.

Dynamic scheduling can be used for all models of computation but the Strict MoC. The Strict MoC is unable to use dynamic scheduling because dynamic scheduling does not guarantee that a W-codeblock executes at most once per cycle in situations like that of Figure 3.1(a). Dynamic scheduling is appropriate for use with the HSR MoC – the least fixed point of the system of codeblock equations can be found by simply invoking each W-codeblock when an input signal changes – but as will described in Section 3.5.4 there are some corner cases involving constant signal values that require modifications to the basic scheduling algorithm.

VeriSUIF's technique could be used for all models of computation, with the possible exception of the Strict MoC. Whether the state machine VeriSUIF generates would guarantee the strictness property (or even whether such a property would have meaning if the code is being analyzed to such a fine granularity) is unclear, but the use of levelization in picking the next state might cause strictness to hold. The VeriSUIF technique does require an enormous state space when a codeblock has multiple outputs. Thus the technique is likely inappropriate for large-scale microarchitectural simulation.

The remaining four techniques assume zero delay and thus cannot be used for the General Discrete Event MoC. They have striking similarities: each technique recognizes the importance of cycles in either the codeblock graph or the signal graph and provides a means to deal with them. For the HSR MoC, this is enough; all four techniques can support this MoC, though both levelized event-driven and acyclic scheduling, as they have components of dynamic scheduling, require the same modifications to handle constant signals that pure dynamic scheduling does.

While it would seem that as a simpler MoC, the Strict MoC could be scheduled using any of the four cycle-based techniques, the strictness property can create problems. Hamiltonian subschedules and levelized event-driven scheduling will work for the Strict MoC as these techniques as introduced operate on the codeblock graph. Because a codeblock can appear at only one level in a strict model, the levelization inherent in these techniques will guarantee strictness. Acyclic scheduling will not work for this MoC because it uses a signal graph and in situations like that of Figure 3.1(b), the tlevel-based invocation coalescing will not guarantee strictness. Partitioned scheduling as introduced will also not work in situations like that which will be characterized in Section 3.5.3. However, a novel coalescing technique will be introduced in that section which would allow both acyclic scheduling and partitioned scheduling to be used. Of course, given the simplicity of standard levelization of the codeblock graph, using any more complex technique for scheduling a Strict MoC model would be pointless.

The applicability of partitioned scheduling to the zero-delay DE MoC is a very interesting question, as partitioned scheduling creates a completely static schedule which may have performance benefits. However, the analysis of this applicability requires a more extensive explanation, which is given in the next subsection.

3.4.1 Applying Partitioned Scheduling to the Zero-Delay DE MoC

Partitioned scheduling produces correct signal values for the zero-delay DE MoC whenever the model is microarchitecturally synchronous. This is not surprising as the HSR MoC for which partitioned scheduling was developed can handle most microarchitecturally synchronous models, but I prove here that partitioned scheduling will indeed produce correct results in the zero-delay DE MoC.

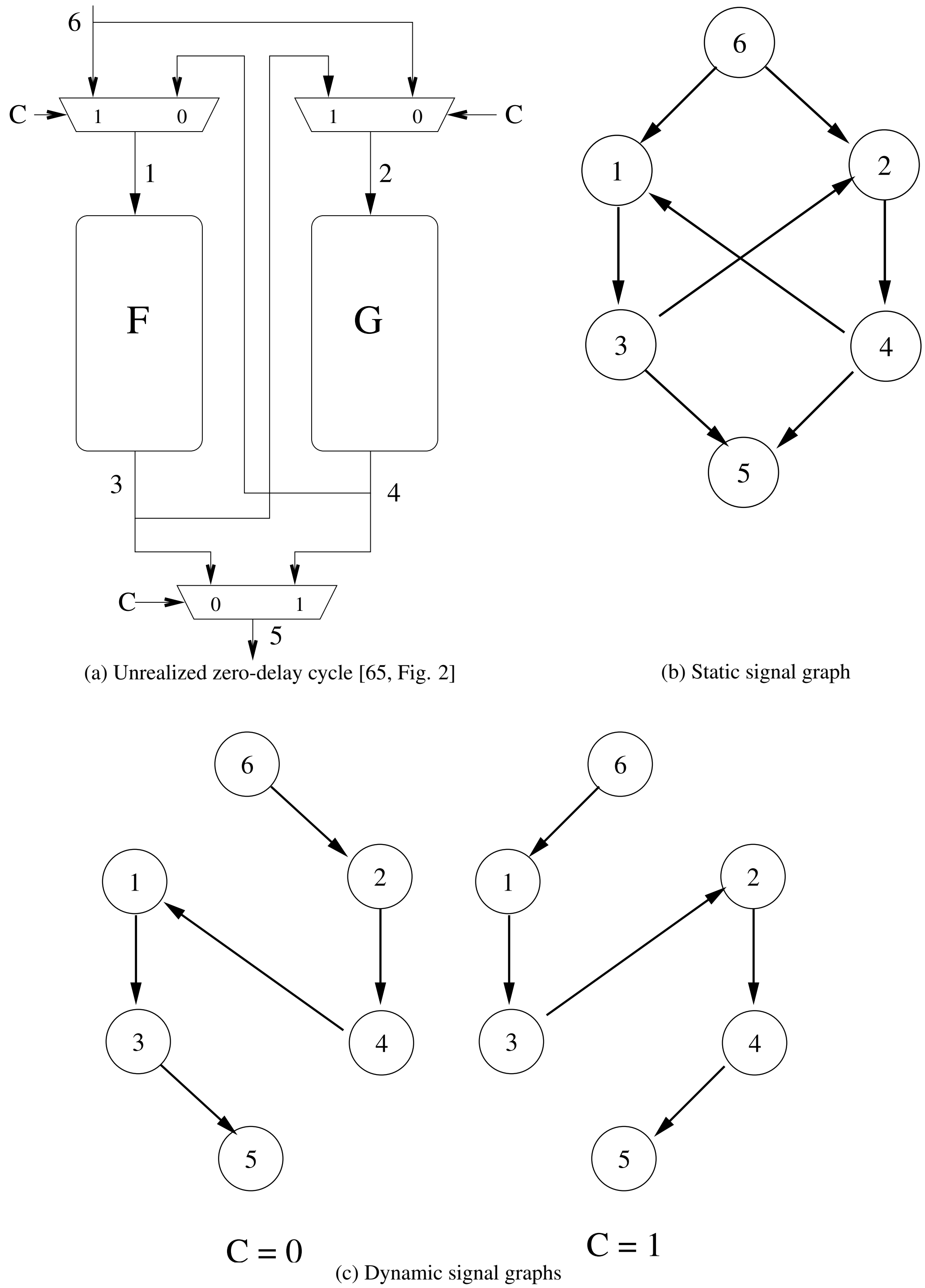
This proof requires a definition of the **dynamic signal graph**. Signal graphs as previously defined are static; if a computational dependence may ever exist between two signals, there is an edge between their vertices in the graph. A dynamic signal graph is a subgraph of the static signal graph with edges removed dynamically as state internal to the modules and signal values change, causing some edges to not be realized. A new instance of the dynamic signal graph exists for each timestep of simulation. Figure 3.7(a), which repeats Figure 2.3(b), gives an example of a model with a dynamic signal graph which can change depending upon a signal value. Figure 3.7(b) shows the static signal graph and Figure 3.7(c) shows the two different dynamic signal graphs.

Note that only edges which are realized in a timestep are included in dynamic signal graph for that timestep. Because of this, emergent state exists at a particular timestep if and only if the dynamic signal graph at that timestep contains a cycle and an output signal or new state value is reachable from that cycle. This implies that if a model is synchronous, any cycles in dynamic signal graphs must not affect the primary outputs or new state values.

The following lemma states a relationship between partitioned schedules and the signal graph:

Lemma 1. *If there exists a path between signals u and v in (static) signal graph G, the value for signal v will be evaluated after the value for signal u at least once in the schedule formed by partitioned scheduling.*

Proof. Suppose that signals u and v are in different strongly-connected components of the signal graph. Either there is a path from u to v or from v to u, but not both. Assume that the path is from u to v. Because components are scheduled in topological order, the component of which v is a part will be scheduled after the component of which u is a part and thus the evaluation of v is scheduled after the evaluation of u.



(a) Unrealized zero-delay cycle [65, Fig. 2]

(b) Static signal graph

(c) Dynamic signal graphs

Figure 3.7: Model with changing dynamic signal graph

Suppose instead that u and v are in the same strongly-connected component. This component will be hierarchically partitioned. There are four cases:

1. Both u and v are in the head of the partition. They will both be scheduled at least twice because the head has at least two signals. Either the evaluation of v will follow that of u in the same iteration or the evaluation of v will follow the evaluation of u from the previous iteration.

2. u is in the head and v is in the tail. The schedule will evaluate signals in the tail, then the head, then again in the tail, followed by possibly more iterations of head and tail. Thus the the evaluation of v in the second iteration of the tail will follow an evaluation of u in the head.

3. v is in the head and u is in the tail. The evaluation of v in the first iteration of the tail will follow an evaluation of u in the head.

4. Both u and v are in the tail. The subgraph represented by the tail is scheduled recursively. By induction on the recursion, v will be evaluated after u in the schedule produced for the tail and thus the evaluation of v will follow that of u in the overall schedule.

□

The following theorem establishes the correctness of partitioned scheduling for microarchitecturally synchronous models:

Theorem 1. *Partitioned scheduling results in schedules which calcuate the correct signal values for zero-delay DE models which are microarchitecturally synchronous.*

Proof. Assume that the model is microarchitecturally synchronous. Then no output signal or new state value ever depends transitively upon an emergent state signal. This is equivalent to saying that there is never a path in the dynamic signal graph which contains a cycle and ends at any output signal or signal which is required to compute a new state value.

Let G be the static signal graph and D_t be the dynamic signal graph at any timestep t. Let D_r be D_t with all cycles as well as any vertices reachable from cycles removed. Because the removed signals have no effect on output signals in this timestep or the next state of the system, the values computed for these signals are irrelevant and an arbitrary scheduling of their computation is sufficient.

For the remaining signals, because D_r is acyclic and a zero-delay model is assumed, a schedule which generates correct signal values is one which obeys topological order: if there exists a path from vertex u to vertex v in D_t then v is scheduled for evaluation after u. Because D_r is a subgraph of G, if there exists such a path in D_r, it also exists in G. By Lemma 1, the evaluation of v will be scheduled after the evaluation of u at least once in the schedule formed by partitioned scheduling for G, therefore this schedule will generate correct signal values for timestep t.

Because D_r is acyclic in all timesteps and signals which affect outputs have never been removed in its formation, the single schedule formed by partitioned scheduling will compute correct signal values in all timesteps, and partitioned scheduling results in schedules which calcuate the correct signal values for zero-delay DE models which are microarchitecturally synchronous. □

The majority of interesting microarchitectural models of synchronous systems are microarchitecturally synchronous; to date all real or proposed systems which we have attempted to model have been so. Nevertheless, extending the applicability of partitioned scheduling to logically synchronous but microarchitecturally asynchronous models, such as that in Figure 2.1 would increase the modeling flexibility offered to microarchitects. Partitioned scheduling can be made to work for these additional models. The key is to break apart the signals into subsignals for analysis in such a way that the dynamic signal graphs become acyclic. For an N-bit signal, there will be at most N such subsignals; by using N subsignals, the model becomes a logic-level model. Partitioned scheduling then can operate upon the subsignals; it can even assume that a subsignal computationally depends upon all the signals which it parent depends upon, though refinement of dependences to the subsignal level through analysis would result in better static schedules.

Breaking apart these signals may lead in many cases to excess invocations, but these can be handled through the dynamic subschedule embedding technique which will be introduced in Section 3.5.2. However partitioned scheduling does quite well at forming schedules for cycles where some signal is a single bit signal. As many apparent cycles in microarchitectural models result from control flow signals which are single bits, partitioned scheduling should do quite well.

Note that the HSR MoC cannot use this technique of breaking apart signals to allow execution of logically synchronous models. Each bit of the signal would need to have an independent possibility of being $\perp$ and the codeblocks would have to be correspondingly modified.

The fact that the bit representation is involved in this adaptation of partitioned scheduling might be a cause for concern. What about models with signal values which are complex data structures? Such models could cause practical difficulties, but as long as the signal can be partitioned into a fixed maximum number of subsignals, the technique will work. The schedule may even still be efficient depending upon the other signals in the cycle. If a fixed number of subsignals cannot be assumed, choosing a number that is so large that dynamic subschedule embedding will come into play will ensure that the scheduler can still be used.

This proof refutes statements made by other researchers about the static scheduling of zero-delay DE models. Gracia Pérez, et al.[36] argued that partitioned scheduling is not appropriate for zero-delay Discrete Event systems such as SystemC because of the difficulty of meeting HSR requirements on codeblocks. The proof shows that partitioned scheduling can be correct for synchronous models without requiring that codeblocks be written to obey an HSR MoC. Jennings argued vehemently[46] that the Discrete Event MoC should not be used at all for digital system design because of the overhead of dynamic scheduling, advocating instead the use of the Strict MoC, while acknowledging that multiple independent outputs per codeblock would not be handled well. The successful creation of static schedules by partitioned scheduling invalidates this argument.

Hamiltonian subschedules and partitioned scheduling have an interesting relationship: they behave precisely the same when there are no cycles in the graph, but handle the schedules for cycles quite differently. In each case, a polynomial-time algorithm to generate the schedule for cycles is not known. However, partitioned scheduling attempts to directly optimize the number of signal evaluations and uses the hierarchical structure of cycles in the signal graph to do. Hamiltonian subscheduling attempts to simply find an order in which to invoke every codeblock in the cycle; it is not even clear that such an order will minimize the number of iterations of the cycle. As partitioned scheduling is a more direct and provably optimal approach, Hamiltonian subscheduling will not be explored further in this dissertation.

3.5 Solving Practical Scheduling Problems

There are several practical issues which must be addressed before scheduling techniques described in the previous section can be applied to microarchitectural simulation. These issues are:

1. Black-box codeblocks do not provide enough information to schedule signals effectively.
2. Partitioned scheduling of models with large cycles suffers from combinatorial explosion.
3. Scheduling signals instead of codeblocks may result in redundant invocations, requiring invocation coalescing.
4. Some W-codeblocks may need to be forced to run in a schedule which is not fully static.

This section describes novel scheduling improvements which address these four issues.

3.5.1 Enhancing Dependence Information

Both partitioned and acyclic scheduling rely upon the signal graph. To form this graph, a knowledge of the computational dependences between signals is necessary. But this knowledge is not directly available when modules or codeblocks are treated as black boxes. As a result, the graph must be assume that all output signals of a codeblock depend upon all the input signals of a codeblock, turning the signal graph into a codeblock graph. The results of this can be poor static schedules where codeblocks are called more frequently than is necessary.

The problem of finding computational dependences is further exacerbated because microarchitectural simulators may not know which codeblock produces a given output signal; indeed, it may be produced through multiple codeblocks. In the complete absence of information, *all* output signals of a component signals must be assumed to depend upon *all* input signals.

In a highly-reusable framework such as LSE which has distributed control logic and black-box modules, these situations occur frequently. In fact, they occur at every connection between modules because of the distributed flow control logic. Thus many extraneous computational dependences are seen. To control this problem, I proposed the following in [75]:

1. Annotations of **port independence**. These indicate that all signals received on a port do not affect any output signals in the same cycle. A port independence annotation is a simple boolean flag on a port declaration in LSE.

2. Addition of **dependence annotations** to modules. These annotations indicate for each output signal the input signals upon which it is computationally dependent. Note that this annotation has finer granularity than that of [78] which only indicates that outputs are dependent upon some input but not which input.

3. Analysis of control functions to determine their computational dependences.

Dependence annotations in LSE are a list of three-tuples giving a source signal name, destination signal name, and an expression indicating which port instances are to be considered "linked". For example, the dependence annotation for a "wire" module (a module that simply passes through signals on each port instance) is:

```
port_dataflow = <<<[
  ('*', '*', '0'),  # eliminate all default dataflow
  ('in.data', 'out.data',  'isporti==osporti'),
  ('in.en', 'out.en', 'isporti==osporti'),
  ('out.ack', 'in.ack',  'isporti==osporti')
 ]>>>;
```

Annotations are not required for correct execution; their presence merely removes unnecessary edges from the signal graph, improving simulation speed. Annotations can be added incrementally to tune the performance of the simulator. Furthermore, annotations need be added only once to a library module, with the cost of doing so amortized over all the uses of the module. It is important, however, that annotations never remove signal graph edges which are true computational dependences. Also, while not currently implemented, it should be possible to automatically extract the annotations from a dataflow analysis of the code of modules; such automatic extraction is left to future work.

The designers of FastSysC[36] have chosen to use similar dependence annotations, citing the results in [75], though in that system they are specified at initialization time with a different syntax. Annotations can be added incrementally in their system and need not be correct; if edges representing true program dependences are removed, then the dynamically scheduled portion of the schedule computes the correct signal values with some performance loss.

The analysis of control functions is done by looking at each return statement in the function to determine what values are being returned. The variable names for input signals are understood; so are signal value manipulation API calls. For example, consider the simple control function which sets a constant acknowledge signal while not modifying other signals:

```
return (LSE_signal_extract_enable(istatus) |
        LSE_signal_extract_data(istatus) |
        LSE_signal_ack);
```

The analysis determines that the output acknowledge signal is a constant and that the data and enable signals are passed through without modification. Note that some experience is needed to write control functions which are easily analyzable.

Dependence information enhancement solves the problem of removing extraneous edges from the signal graph. It does not, however, directly solve the problem of knowing what codeblock may generate an output signal when there are multiple codeblocks in a module. In LSE, this information can be partially inferred because of the classification of W-codeblocks. Output signals can be generated by the phase codeblock or any handler which is tied to an input signal upon which the output signal is dependent.[1]

This knowledge is used further to reduce handler invocations. When an output signal is to be computed, the codeblocks which are invoked are the phase codeblock and the handlers for input signals upon which the output depends. However the handlers are only invoked if the signals which they handle have been computed since the last time the handler was invoked. This complicates the generation of codeblock schedules from signal schedules, but not the generation of signal schedules. For dynamic scheduling, only the appropriate handler is scheduled to run; if there is no handler, the phase codeblock is scheduled.

3.5.2 Dealing with Combinatorial Explosion

As mentioned in Section 3.3.4, a polynomial-time algorithm for finding an optimal static schedule for HSR models is not known. The heuristics presented by Edwards in [21] are still exponential

[1]Control functions can only generate special signals added when a control function is attached.

in the worst-case scenario. Use of a scheduling algorithm which is exponential in the number of signals to be scheduled could severely limit scalability in model size. Thus it is necessary to modify the scheduling algorithm.

When the signal graph is acyclic, the partitioned scheduling algorithm requires linear time: once it finds a the strongly-connected components it simply lists them in topological order. Combinatorial explosion occurs only when there is a cycle in the graph and the algorithm begins to recursively consider partitions of the cycle. These cycles occur for three reasons:

1. There is truly a zero-delay cycle in the computational dependences. Such a situation is likely an error, as the signals involved in the cycle will remain $\perp$ in an HSR model.

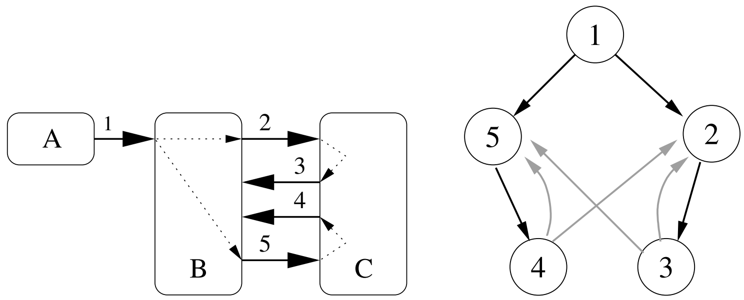
2. A static zero-delay cycle is never realized dynamically, as in Figure 2.3(b).

3. Imprecise information about computational dependencies causes spurious edges to appear in the signal graph, as in Figure 2.3(a).

The final reason occurs frequently; in fact, without dependence information enhancement as described in the previous subsection, it is extremely common. Thus it becomes essential to solve the problem of combinatorial explosion.

The solution involves a subtle observation that when dependence information is missing or dependence changes dynamically, partitioned scheduling is solving the wrong problem. In either case partitioned scheduling provides an optimal schedule for a set of computational dependences which is a superset of the true dependences. This schedule is not optimal for the *true* computational dependences at each timestep. Thus standard partitioned scheduling is not truly optimal!

As a detailed example of how a lack of information compromises optimality of the partitioned scheduling algorithm, consider the codeblock graph in Figure 3.8(a) and its corresponding signal graph in Figure 3.8(b). The dotted lines inside the codeblocks indicate the true computational dependences within the codeblocks blocks. Signals 2 and 5 are computed from signal 1. Signals 3 and 4 are computed from signals 2 and 5 respectively, and signals 3 and 4 are not used to compute any other signals with zero delay. Suppose that the scheduler does not know about these true dependences. It must conservatively assume that all possible dependences within the codeblocks exist. The extra edges which the scheduler must assume are shown as gray lines in the



(a) Codeblock graph

(b) Signal graph

$A\ C\ B\ C\ B\ C$

(c) "Optimal" Schedule

$A\ B\ C$

(d) True Optimal Schedule

Figure 3.8: Non-optimal schedules due to lack of information

signal graph. The "optimal" schedule of invocations (after optimal invocation coalescing) which the scheduler then generates is $A\ C\ B\ C\ B\ C$, yet the schedule $A\ B\ C$ is correct and shorter.

The solution is to *not* attempt to partition and form an optimal schedule for a large SCC. Instead, the evaluation of the signals within the SCC is done using event-driven scheduling. I call this technique **dynamic subschedule embedding**. It is similar in some ways to what Hommais and Pétrot did with SCCs in the codeblock graph, however in this case dynamic scheduling is applied to only the large SCCs of the signal graph. Furthermore, instead of using relaxation and an NP-complete algorithm to find subschedules which will just have to be iterated to convergence, dynamic scheduling is used, thus allowing the order of invocation to be driven by the true computational dependences. Because a hierarchical decomposition of large SCCs is not attempted, the time complexity of scheduling becomes $\Theta(V_S + E_S)$.

Choosing the size at which SCCs become large is important; for very small SCCs the additional overhead of dynamic scheduling may be larger than the cost of extra codeblock invocations. For this work the definition of a large SCC is set by the interaction of both the initial scheduling pass which produces a hierarchical schedule and an "unrolling" pass which produces a linear schedule. The rules for the partitioning of SCCs are:

1. If $|SCC| \geq N_{large}$, insert a dynamic section in the first pass. This rule provides the bound on the running time of the algorithm.

2. If $N_{small} \leq |SCC| < N_{large}$, attempt to schedule recursively with a single partition where the head is the border of the set of signals in the SCC with the same driving codeblock as

the first vertex in the SCC. The purpose of this rule is to give medium-size SCCs such as might arise from Figure 2.3(b) some opportunity to be statically scheduled.

3. If $|SCC| < N_{small}$, schedule recursively, exhaustively trying all possible partitions. This rule generates an optimal schedule (with respect to the signal graph) for very small SCCs.

4. If a repeating portion of the schedule has more than 2 repetitions and its total cost will be greater than N_{small}, replace it with a dynamic section in the second pass. This rule filters out subschedules generated in step 2 which have large numbers of repetitions.

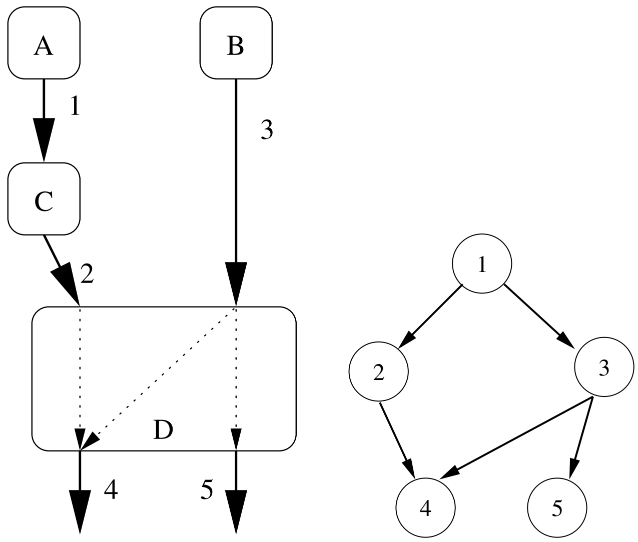
N_{large} and N_{small} are tuneable parameters of the partitioning algorithm.

It is interesting to contrast dynamic subschedule embedding with acyclic and levelized event-driven scheduling. With dynamic subschedule embedding the philosophy is to "contain" the dynamic behavior of the system to just those signals involved in large cycles. Levelized event-driven scheduling contains dynamic behavior of small cycles within subschedules, but still leaves the global event-driven scheduling to be dynamic in nature. Acyclic scheduling does not contain dynamic behavior; any cycles in the system will force all signal evaluations "downstream" of the cycle to be triggered dynamically.

Note that Edwards' concern when introducing the HSR model of computation was to define a system which allowed an optimal static schedule for the worst-case execution of a software system. Large scheduling times, while unpleasant, were acceptable, and the systems envisioned had at most hundreds of signals. The concern here is for systems that may have thousands or even tens of thousands of signals with users sensitive to simulator build times. Furthermore, the simulator does not require a worst-case execution time bound; good average-time behavior is sufficient. Thus, a solution with dynamic behavior is acceptable.

3.5.3 Coalescing Invocations

Techniques such as partitioned scheduling and acyclic scheduling use the signal graph rather than the codeblock graph. This provides better schedules but at the price of potentially redundant codeblock invocations when codeblocks produce multiple signals. Both techniques handle this by coalescing invocations.



(a) Codeblock graph

(b) Signal graph

$B(3)\ D(5)\ A(1)\ C(2)\ D(4)$

(c) Initial Schedule

$B(3)\ A(1)\ D(5)\ C(2)\ D(4)$

(d) Tlevel-based

$B(3)\ D(5)\ A(1)\ C(2)\ D(4)$

(e) Signal-based

$B(3)\ A(1)\ C(2)\ D(4,5)$

(f) An Optimal Coalescing

Figure 3.9: Limitations of coalescing techniques

Acyclic scheduling uses a tlevel-based approach: if two invocations are to generate signals which are at the same *tlevel* in the signal graph after back edges have been removed, they may be coalesced. Partitioned scheduling uses a signal-based approach; signal evaluations are moved earlier in the schedule until they can be coalesced with other evaluations by the same codeblock.

Both of these techniques are helpful, yet there are situations where each fails. Figure 3.9 shows a codeblock graph and signal graph for which neither technique achieves an optimal coalescing. Internal dependences have been given to the scheduler through dependence annotations. The initial schedule is that computed using partitioned scheduling where the signal vertices are initially considered in numeric order while forming the SCCs; both codeblocks and signal numbers are shown. Tlevel-based coalescing reorders this schedule, but cannot coalesce. The underlying problem with tlevel-based coalescing is that when output signals of a codeblock use input signals which come from different parts of the model, the *tlevel* of the output signals is often different and coalescing does not occur. Signal-based rescheduling does not have this problem, but is limited by the fact that single signals are moved at a time. In this particular example the second invocation of codeblock D cannot be coalesced because it is dependent upon invocations of codeblocks A and C, which are not moved because they have only a single invocation each.

I propose the use of a novel subgraph-based technique for invocation coalescing for partitioned scheduling[75]. This new technique moves the evaluation of sets of signals which form an acyclic subgraph. The algorithm is given in Figure 3.10. The idea is to move not just a single signal evaluation, but also the signal evaluations upon which it depends. This technique is able to produce the optimal coalescing for the example in Figure 3.9. The worst-case running time for this algorithm is $O(V_S^2(V_S + E_S))$.

SUBGRAPH-COALESCING(S)

▷ S is the original schedule: a list of invocations records which have three fields:
  $codeblock$ indicating the codeblock to invoke
  $signalList$ indicating the set of signals being produced by the invocation.
  $depList$ indicating $\bigcup_{v \in signalList}$ signals upon which v depends
 Signal lists initially contain a single signal.

$earliest \leftarrow$ EMPTY   ▷ $earliest$ is a mapping from codeblock to position
foreach invocation s **in** S
  do
   if $s.codeblock$ does not have a position u in $earliest$
    then Store mapping from $s.codeblock$ to current position in $earliest$
    else $H \leftarrow$ NIL
      $deeps \leftarrow s.depList$
      foreach invocation t **in** S s.t. t is after u but before s,
          moving backwards through S
       do
        if $\exists v \in t.signalList$ s.t. $v \in deps$
         then if $t.codeblock = s.codeblock$
            then Exit inner **do** loop without coalescing
            else Prepend t to H
              $deps \leftarrow deps \bigcup t.depList$
        elseif $t.codeblock = s.codeblock$
         then ▷ Coalesce s and t
           $t.signalList \leftarrow t.signalList \bigcup s.signalList$
           $t.depList \leftarrow t.depList \bigcup s.depList$
           Remove s from S
           Insert H into S before t
           Exit inner **do** loop

Figure 3.10: Subgraph coalescing

Each of the techniques can be considered a heuristic for finding an optimal coalescing. Finding this optimal coalescing (or more precisely, the decision problem of whether a coalescing which meets a bound on the number of invocations exists) has been shown to be NP-hard by Wang and Maurer[105] by reduction from the Shortest Common Supersequence problem.

Subgraph-based coalescing makes it possible to use either acyclic scheduling or partitioned scheduling for the Strict MoC. In the Strict MoC, both the codeblock graph and the signal graph are acyclic. Because the codeblock graph is acyclic, there are no situations where the inner loop has to exit without coalescing and all invocations of the same codeblock can be coalesced, leading to at most one invocation per codeblock, thus satisfying strictness.

3.5.4 Forced Invocation

As mentioned before, dynamic scheduling can be used for an HSR model only with some improvements. When a W-codeblock invocation is statically scheduled, it can be assumed that it will be invoked. However, when an invocation is dynamically scheduled or may be skipped due to selective-trace scheduling, this assumption cannot be made and there is sometimes a need to "promote" the invocation from dynamic to static or to suppress skipping if the normal process of scheduling when a signal value is assigned does not occur. This happens in three situations:

1. The output signal depends solely on constant input signals and internal state. Constant input signals may be inferred from unconnected port instances or from control function analysis. A W-codeblock with no input signals such as the phase start function in LSE would be a special case of this situation.

2. For control functions in LSE, control function analysis cannot determine whether the output is a constant. If it may be a constant, the codeblock must be run to produce that constant in case that assignment is needed to cause other codeblocks to execute.

3. For handlers in LSE, if the handler is for an unconnected port instance, the handler must be forced to run.

The scheduler I created handles this issue by looking at the first scheduled invocation of any codeblock which needs to be forced. If that invocation is static and selective-trace scheduling is enabled, then the invocation is flagged as requiring forcing. If the first invocation is dynamic, it is added to a list of forced invocations for the embedded dynamic subschedule. How this list is used depends upon the scheduling technique.

3.6 Evaluation of Static and Hybrid Scheduling Techniques

This section presents an evaluation of the effectiveness of static and hybrid scheduling techniques for microarchitectural simulation. The goal is to determine which techniques are best suited for this context. The three techniques evaluated are acyclic scheduling, levelized event-driven scheduling, and partitioned scheduling. Where appropriate, selective-trace scheduling is evaluated with the main scheduling technique. In addition, the novel dependence information enhancement mechanisms and subgraph coalescing proposed in Section 3.5 are evaluated.

3.6.1 Evaluation Methodology

All evaluations are carried out in the Liberty Simulation Environment. Except where otherwise noted, all dependence information enhancement mechanisms are used. Each scheduling technique is evaluated separately against a baseline of dynamic scheduling. The best variants of each technique are then compared in Section 3.6.5

Models

The evaluation uses six different processor models. Three of these models have been used in published work to evaluate microarchitectural techniques. The framework is used to generate multiple simulators using the different scheduling techniques. The performance of each of these simulators is measured as well as the number of codeblocks invocations per codeblock running different benchmarks and input sets. Performance measurements are taken from one simulation run; simulation lengths were chosen to achieve wall-clock time of at least 15 minutes to reduce system effects. For all runs the reported CPU time was greater than 99% of wall-clock time.

The models are summarized in Table 3.2. The table indicates the number of module instances, signals, and codeblocks for each model. Each of the models and their input sets is described in more detail below:

Model	Instances	Signals	Codeblocks	Benchmarks
ALPHA	46	1305	515	3 SPEC INT 2000
CMP04	312	4182	1555	3 Splash2 kernels
I2	209	3869	1508	3 SPEC INT 2000
I2-CMP	530	13411	5177	3 Splash2 kernels
PPC	144	4017	1749	3 SPEC INT 2000
RISC	49	407	94	2 kernels

Table 3.2: Models and input sets

ALPHA

ALPHA is a structural model of an out-of-order processor executing the Alpha ISA. The author is David A. Penry. The microarchitecture is intended to match exactly that of the processor modeled in the popular SimpleScalar[12] `sim-outorder` simulator. The simulator matches the output of `sim-outorder` precisely. An early version of this model was used in a tutorial on LSE at MICRO-34. With full dependence information enhancement, the signal graph is acyclic, but the codeblock graph is not.

Benchmark	Input	Starting instruction	Length
256.bzip2	image	71,900,000,000	150,000,000
164.gzip	program	119,000,000,000	150,000,000
186.crafty	reference	77,500,000,000	150,000,000

Table 3.3: Input and sampling parameters for ***ALPHA***

Three benchmark and reference input combinations from the SPEC CPU 2000[1] benchmark suite are used as shown in Table 3.3. These particular benchmarks and input set combinations were selected because they provide the lowest, median, and highest slowdown respectively between the structural simulator and `sim-outorder`. The binaries were downloaded from http://www.simplescalar.com; the compiler used to create these binaries is not known. For each benchmark, the beginning of the single SimPoint[84] listed on the SimPoint website is used as the starting point of execution; 150 million instructions are executed. For the three benchmarks and input combinations shown here, the dynamically scheduled simulator generated from this model is on average 23 times slower than `sim-outorder`, which is a highly-optimized hand-coded simulator written in C.

Benchmark	Input parameters	Starting instruction	Length
cholesky	-p4 -C32768 lshp.O	30,000,000	10,000,000
fft	-p4 -m16 -n1024 -l5	30,000,000	10,000,000
radix	-p4 -n262144	20,000,000	10,000,000

Table 3.4: Input and sampling parameters for ***CMP04*** model

CMP04

CMP04 is a model of a chip multiprocessor with four processors. The authors are David A. Penry and Julia S. Chen. The microarchitecture is tiled, i.e. there are four tiles, each containing a processor with first level instruction and data caches, a portion of a distributed second-level cache, and connections to an on-chip routing network. Each tile has an independent channel to memory. The processors are derived from the ***ALPHA*** model, but are configured to be scalar and in-order and to use the PowerPC instruction set. This model is very similar to the models used in [16]. With full dependence information enhancement, the signal graph is acyclic, but the codeblock graph is not.

Three combinations of benchmark and inputs from the SPLASH-2 benchmark suite[110] are used. The compiler was gcc 3.4.1 with flags `-g -O2 -lm -static`. A sampling technique is used: one slice of execution is run. This slice begins after some number of instructions have completed on the first core. This number is chosen so that all simulated threads have begun execution by that time. The slice ends when the first core has completed a further fixed number of instructions. The input and sampling parameters are given in Table 3.4.

I2

I2 is a detailed model of the Intel Itanium® 2 processor. The author is David A. Penry. This model has been validated against hardware and is accurate to within 6% absolute average error for the SPEC CPU INT 2000 benchmarks. It has been described in detail in [77]. With dependence information enhancement, the signal graph is not acyclic: there are two signals involved in a cycle. This cycle occurs because one control function has port instance-specific dataflow which cannot be parsed by the framework. The codeblock graph is not acyclic.

Benchmark	Input parameters	Starting instruction	Length
cholesky	-p2 -C32768 lshp.O	50,000,000	500,000
fft	-p2 -m16 -n1024 -l5	40,000,000	500,000
radix	-p2 -n262144	1,000,000	500,000

Table 3.5: Input and sampling parameters for ***I2-CMP*** model

The benchmarks are bzip2, crafty, and bzip from the SPEC CPU 2000 suite. These were chosen simply because these were the benchmarks used for the ***ALPHA*** model. The same inputs are used as were used for the ***ALPHA*** model. The binaries were compiled using ecc 6.0 Beta, build 20011119 with flags `-O2 -static`. The same inputs are used that were used for that model. Sampling is SimPoint-like, except than instead of calculating the SimPoint, the 10 billionth instruction is used as the first instruction. The length of the sample is 15 million instructions.

I2-CMP

I2-CMP is a two-way chip multiprocessor based upon the Itanium® 2 processor model described previously. The author is Ram Rangan. The cache hierarchy is extensively modified and special hardware communication structures have been added. This model was used in [71]. It is the largest of the models, having over 13,000 signals and 5000 codeblocks. The signal graph is not acyclic after dependence information enhancement; there are three cycles involving ten signals. Two of these come from the cycle in the ***I2*** model; the other is due to a control function in a bus arbiter which cannot be parsed. The codeblock graph is not acyclic.

Three combinations of benchmark and inputs from the SPLASH-2 benchmark suite[110] are used. The binaries were compiled using gcc 2.96 with flags `-g -O2 -lm -static`. The same sampling technique used for the ***CMP04*** model is used. The input parameters and beginning of each slice used are given in Table 3.5.

PPC

PPC is a model of an out-of-order processor based loosely upon the PowerPC 970. The author is Ram Rangan. The dependence annotation is incomplete; while it is sufficient to allow partitioned scheduling to create a fully static schedule, there are still many small cycles in the signal graph. In all, there are 26 cycles involving a total of 164 signals. Most of these cycles occur because of control functions written in an unparseable fashion; they have not been rewritten so that the effects of many small cycles – a likely state of affairs for un-performance-optimized models – will be seen. The codeblock graph is cyclic.

The benchmarks are bzip2, crafty, and bzip from the SPEC CPU 2000 suite. The binaries were compiled using gcc 3.4.1 with flags `-g -O2 -lm -static`. These were chosen simply because these were the benchmarks used for the ***ALPHA*** model. The same inputs are used that were used for that model. Sampling is SimPoint-like, except than instead of calculating the SimPoint, the 10 billionth instruction is used as the first instruction. The length of the sample is 250,000 instructions.

RISC

RISC is a model of a simple single-issue in-order processor executing a subset of the MIPS instruction set. It is based upon Figure 6.51 of [74]. This model comes from [36] and was originally written in SystemC by Gilles Mouchard. For these experiments, it has been ported into LSE by David A. Penry. After dependence annotation, the signal graph is acyclic. Because all flow-control signals signals are forced to be constants in this model, the codeblock graph is acyclic as well. This is a very small model, but it is interesting because one can expect all static scheduling techniques to produce exactly one codeblock invocation per cycle, allowing comparison of runtime overheads of different techniques.

The benchmarks used in [36] were not specified and the authors are now unable to reproduce the benchmarks[35]. In addition, the subsetting of the MIPS instruction set makes it difficult to write large benchmarks. As a result, two small assembly-language kernels are used. The first repeatedly calculates the millionth Fibonacci number using an iterative algorithm. The second repeatedly performs a 50 by 50 matrix multiplication. In each case 500 million instructions are simulated.

Implementation of Dynamic Scheduling

The framework uses a straightforward single-pass dynamic scheduling algorithm. A list of codeblocks to be invoked is maintained. This list is traversed to invoke the blocks. When a signal is

$\triangleright$ $SchedList$ is a FIFO queue.
$\triangleright$ $flags$ is an array of codeblock flags.
$\triangleright$ $ForceList$ is the list of codeblocks which must be forced to execute

```
DYNAMIC-SCHEDULE(C)
   ▷ C is the codeblock to schedule.
   if flags[C] = FALSE
      then flags[C] ← TRUE
           ENQUEUE(SchedList, C)
```

```
DYNAMIC-INVOKE()
   foreach codeblock C in ForceList
        do DYNAMIC-SCHEDULE(C)

   while SchedList not EMPTY
        do C ← DEQUEUE(SchedList)
           flags[C] ← FALSE
           Invoke C
```

Figure 3.11: Dynamic scheduling operations

given a value, all codeblocks which receive the signal as inputs are appended to the end of the list if they are not already present. Figure 3.11 shows pseudocode for the scheduling and invocation operations.

Compilation and Evaluation Systems

All simulators were compiled using gcc 3.4.4 with the default compilation flags provided by LSE's `ls-build` script. All simulations were run on a single processor system with one AMD Athlon™64 Processor 3400+ at 2.4 GHz. This system has 512 kilobytes of L2 cache and 2 gigabytes of memory. The system was running Fedora Core release 3 with kernel version 2.6.9-1.667smp.

Wall-clock time is measured using `/usr/bin/time`. The wall-clock time includes time to start the simulation binary, initialize, finalize, and perform all I/O including reading of simulation state checkpoints. A statistical option in LSE measures the number of times each codeblock is invoked.

3.6.2 Acyclic Scheduling

The acyclic scheduling algorithm was ported from FastSysC[36] into the LSE framework. There is one significant difference; the order in which signals are considered when performing scheduling steps such as depth-first search or topological sort is different from what it would be in FastSysC. In FastSysC, this order is the order in which signal constructors are called during initialization. In LSE the order is complex, but is approximately alphabetical by the module instance and port name driving the signal. As a result, LSE and the original acyclic scheduling may assign different *tlevel*s to signals which are part of a cycle in the signal graph.

The generated code is also somewhat different from that produced in FastSysC because of differences in the MoC and the framework. The generated code for scheduling and invocation operations is shown in Figure 3.12. There are four significant differences:

1. FastSysC uses a two-pass dynamic execution strategy where a change in a signal value does not immediately add codeblocks which read that signal to the invocation list. In contrast, LSE uses a one-pass strategy where codeblocks are immediately added to the invocation list.

2. FastSysC maintains a count of codeblocks which should be invoked at each *tlevel* and can skip an entire *tlevel* or even the entire remaining static schedule if it determines that no codeblocks remain to be invoked. LSE is unable to do this.

3. FastSysC generates individual statements for the testing of flags and invocation of each codeblock in the static schedule rather than generating a table of codeblocks to be executed. Gracia Pérez et al. used this approach for FastSysC because it provides better performance for the small models which they tested; they attribute this performance benefit to improved branch predictor behavior when selective-trace is used. However, such a technique is *not* scalable: in a very large model, both the instruction cache and the branch predictor would be overwhelmed by the static schedule code. For this reason LSE uses a table-driven approach.

4. Due to its DE MoC, FastSysC can compare signal values across clock cycles. LSE cannot do this because the HSR MoC requires all signal values to return to $\perp$ at the beginning

```
▷ StaticSched is the static schedule; each element has fields:
   codeblock: the codeblock to invoke
   force: should invocation be forced?
▷ SchedList is a FIFO queue.
▷ flags is an array of codeblock flags with values TRUE, FALSE, and NEEDED
▷ in-acyclic-finish is a boolean flag


ACYCLIC-SCHEDULE(C)
   ▷ C is the codeblock to schedule
   if in-acyclic-finish = TRUE
     then if flags[C] = FALSE
              then flags[C] ← TRUE
                   ENQUEUE(SchedList, C)
     else  flags[C] ← NEEDED


ACYCLIC-INVOKE()
   ▷ Run static portion of schedule.
   in-acyclic-finish ← FALSE
   foreach E in StaticSched
       do C ← E.codeblock
          if not selective-trace-enabled or E.force = TRUE or flags[C] ≠ FALSE
            then flags[C] ← FALSE
                 Invoke C

   ▷ Run transition portion of schedule.
   in-acyclic-finish ← TRUE
   foreach codeblock C
       do if flags[C] = NEEDED
            then flags[C] ← FALSE
                 Invoke C

   ▷ Run dynamic portion of schedule.
   while SchedList not empty
       do C ← DEQUEUE(SchedList)
          flags[C] ← FALSE
          Invoke C
```

Figure 3.12: Acyclic scheduling operations

of each timestep. Thus FastSysC reports fewer changes in signal values than LSE does. This results in LSE potentially making more codeblock invocations; LSE will invoke each codeblock at least once per cycle.

One should expect to see that acyclic scheduling results in fewer codeblock invocations than dynamic scheduling. This expectation is qualified by the fact that invocation coalescing (which is implicit in dynamic scheduling) may proceed differently even for an acyclic signal graph. Improve simulation speed should come about through two factors: the reduction in the number of invocations and a reduction in the average cost of scheduling a codeblock.

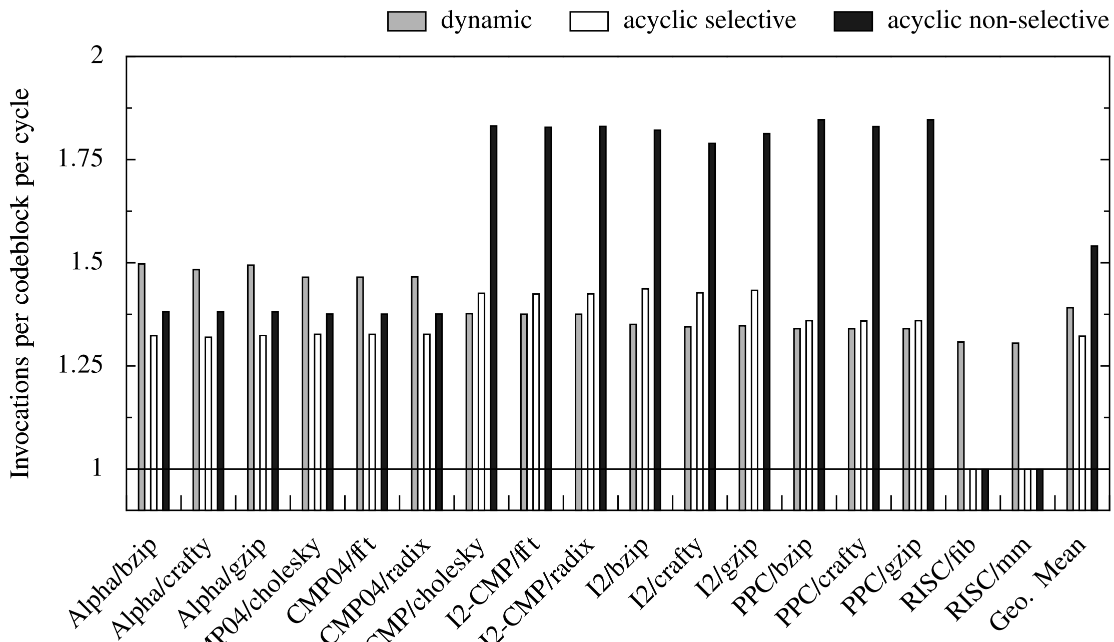
Figure 3.13(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination for dynamic scheduling, acyclic scheduling without selective-trace, and acyclic scheduling with selective-trace. Figure 3.13(b) shows the speedup of each of these three techniques vs. dynamic scheduling. Averages are on the right-hand side of the graph and are geometric means across all models and benchmarks.

These results appear to segment the models into two classes: those that have more codeblock invocations and experience slowdown when acyclic scheduling is used (***I2***, ***I2-CMP***, and ***PPC***) and those that have less invocations and experience speedup.

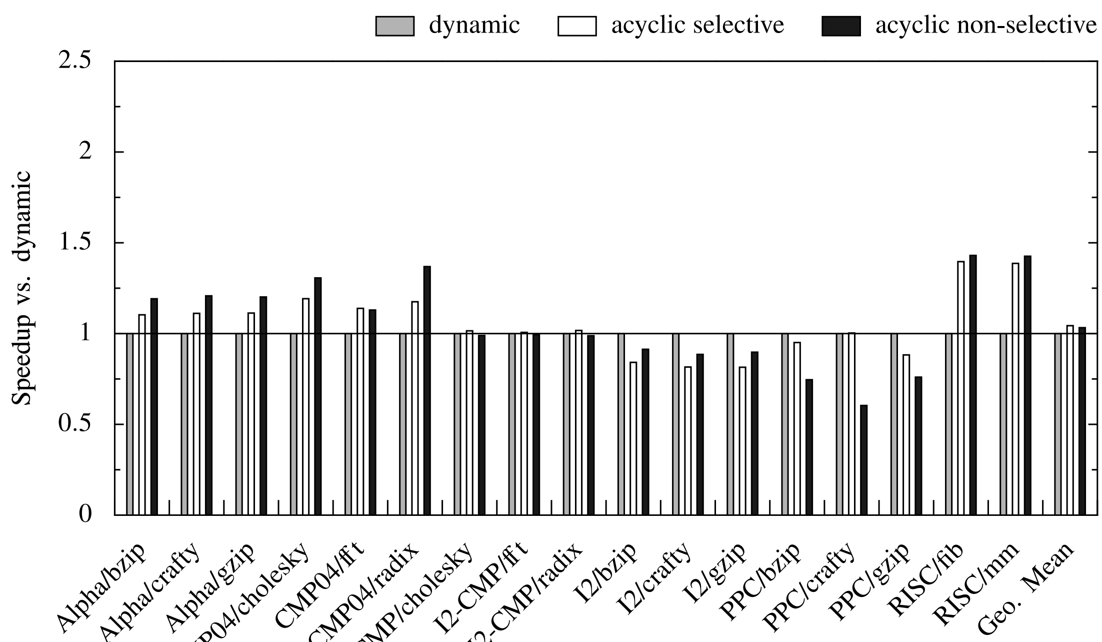
Acyclic scheduling is successful at reducing the number of codeblock invocations for the larger class of models. For the ***RISC*** model (which was used to evaluate acyclic scheduling when it was first proposed) it is able to reduce them to the minimum of one invocation per codeblock. This is due to the fact that the codeblock graph is also acyclic. The reduction in invocations leads to faster simulation speed. These results confirm and extend the results of [36] to the HSR MoC – acyclic scheduling can be an effective technique for improving simulation speed.

Clearly selective-trace scheduling reduces the number of codeblock invocations for these models. But this reduction comes at a price: the extra checks to see whether a codeblock needs to be invoked appear to outweigh the time savings from skipping the invocations unless many invocations are skipped. These results again conform with those of [36].

Note that a direct quantitative comparison of speedup of acyclic scheduling with that given in [36] would be inappropriate even for the ***RISC*** model because the input benchmarks are different, the style of dynamic scheduling being compared against (and thus the speed and number



(a) Invocations per codeblock per cycle



(b) Speedup vs. dynamic

Figure 3.13: Acyclic scheduling results

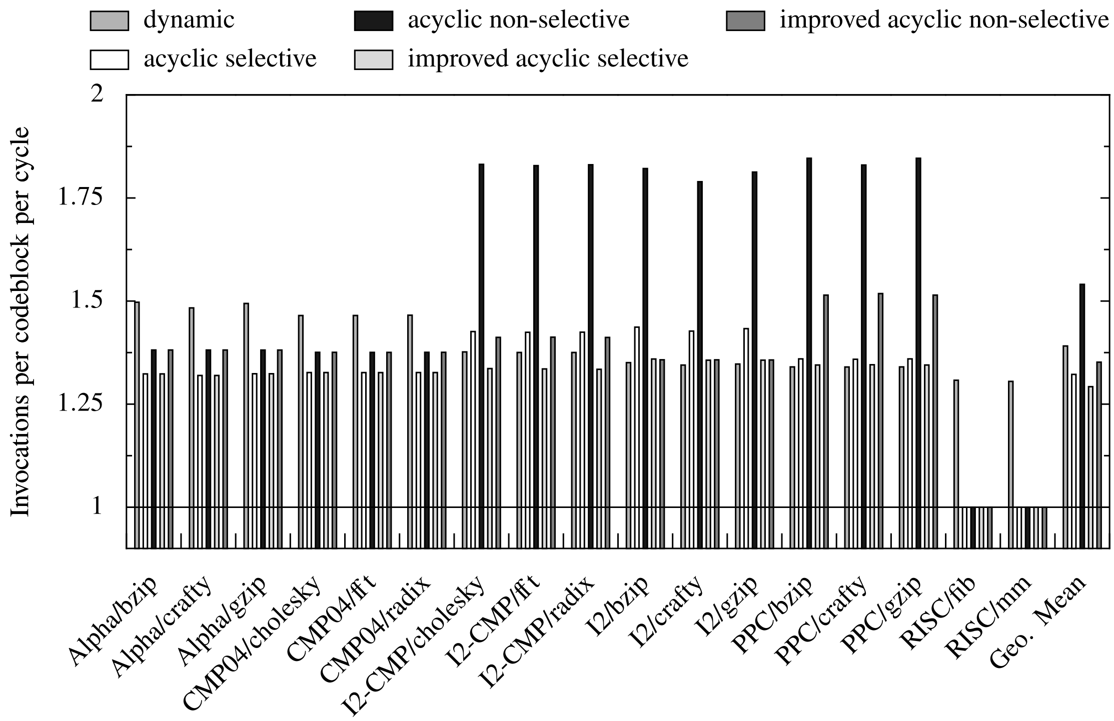
of codeblock invocations for dynamic simulation) is different, and LSE cannot compare signal values across cycles to reduce invocations.

The other class of models shows three unusual tendencies. First, the number of codeblock invocations increases with acyclic scheduling. Second, the number of codeblock invocations soars when selective-trace scheduling is not used. Finally, acyclic scheduling slows down the simulator.

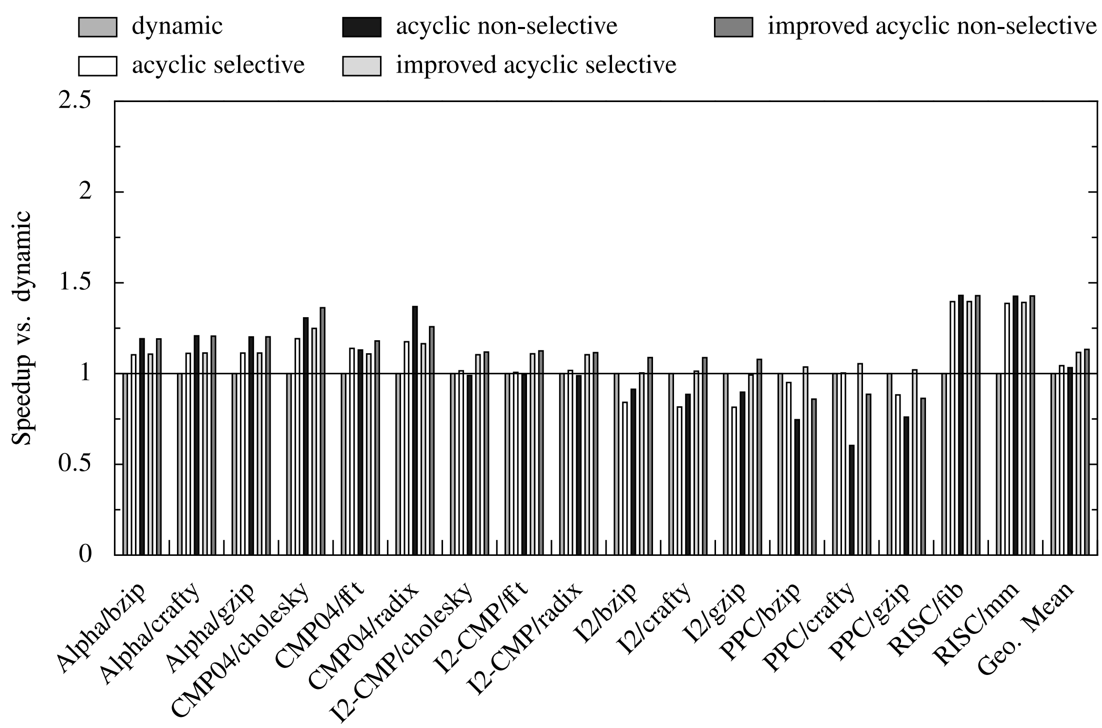
Why do these models behave differently? One important difference between these models and the others is that their signal graphs are cyclic. This implies that the edge removal algorithm comes into play; this algorithm was not required to schedule the models used in [36]. Ideally, parts of the signal dataflow graph which are not involved in cycles should still maintain dataflow order in the schedule so that when they are invoked their input signals are at their final value for the timestep. Analysis of the edge removal code shows that this code does not guarantee this property because it removes edges which are not involved in cycles. The result is that the static portion of the acyclic schedule invokes codeblocks for which the input signals are not yet at a non-$\perp$ value. This effect is transitive as $\perp$ inputs to one level of codeblocks lead to $\perp$ inputs at the next level. Eventually all of these codeblocks are reinvoked during the fallback dynamic scheduling. Selective-trace scheduling prevents the extra static invocations. However, because many of the invocations are deferred until the fallback dynamic scheduling, performance improvement versus the baseline dynamic scheduling is not seen; indeed, the additional overhead of managing the largely-ignored static schedule causes the performance to be less than that of the dynamic scheduling for the ***I2*** and ***PPC*** models.

The edge removal can be improved by removing only cycle-causing edges in the signal graph. These edges are the back edges with respect to the strongly-connected component decomposition of the graph and can be found by using FIND-SCCS-AND-BACK-EDGES. Note that any execution downstream of a cycle will still have to be dynamically scheduled if the edges which are removed represent true computational dependences.

Figure 3.14 shows the results obtained by using the original and improved edge removal algorithms. For the three models with cycles in the signal graph, the improved edge removal algorithm successfully reduces the number of codeblock invocations to levels close to those obtained



(a) Invocations per codeblock per cycle



(b) Speedup vs. dynamic

Figure 3.14: Improved acyclic scheduling results

through dynamic scheduling. The ***I2-CMP*** model joins the class of models with both reduced invocations and speedup, though selective-trace is required to reduce the number of invocations below dynamic scheduling. In fact, some true edges were removed and some dynamic scheduling was required. The ***I2*** model is also able to obtain speedup even though the number of invocations is slightly larger than with dynamic scheduling. The single edge which was removed was a false edge, so the minor difference in the number of invocations is entirely due to differences in invocation coalescing.

On the other hand, the ***PPC*** model shows a unique behavior: selective-trace scheduling improves performance, turning a slowdown into marginal speedup. This model is unique in that it has many small cycles in the signal graph. The edge removal algorithm removes false edges from some of these cycles and true edges from the others. Enough true edges are removed to make selective-trace faster than non-selective-trace, even with the increased invocation overhead. At the same time, enough of the schedule remains statically scheduled to provide overall speedup.

Overall, once the edge removal algorithm is improved, acyclic scheduling is able to provide a speedup versus dynamic scheduling varying from 1.03 to 1.43, with a geometric mean of 1.14. Whether selective-trace is desirable depends upon the model.

3.6.3 Levelized Event-driven Scheduling

The levelized event-driven scheduling technique introduced in LECSIM requires a major modification to be used with microarchitectural simulation. This modification is to schedule using the signal graph rather than the codeblock graph. This is important because in LSE flow control signals cause cycles in the codeblock graph which are not present in the signal graph. However, doing so implies that a codeblock has multiple *tlevel*s: up to one for each of its output signals. Thus a question arises: at what *tlevel* a codeblock should be scheduled when a particular input signal transitions?

The approach which this implementation takes is to compare *tlevel*s for signals and codeblocks. The *tlevel* for each signal is computed using the signal graph. For each codeblock, a **level-set** is formed. This level-set is the set of *tlevel*s of all signals which the codeblock may generate. The level to use for scheduling when a signal changes values is the smallest element of the level-set of the codeblock receiving the signal which is greater than the signal's *tlevel*. If no such element exists, the level used is the smallest element of the level-set.

This solution allows a codeblock to be scheduled at different levels when its output signals depend upon input signals at different *tlevel*s. This also implies that opportunities may exist for invocation coalescing by reducing the size of the codeblock level sets. Tlevel-based coalescing is already present implicitly, as multiple invocations of the same codeblock to generate signals at the same *tlevel* cannot occur because a codeblock can only be scheduled once per level. However, subgraph-based coalescing could also be used and its effectiveness is evaluated here.

For this evaluation, $smallSize$, or the maximum number of signals in a "small" strongly-connected component, is set to either 1 or 15. When it is set to 1, all SCCs are treated as large SCCs and there are no embedded subschedules. When it is set to 15, embedded subschedules may be formed. These SCCs do not use normal invocation coalescing; instead, if a codeblock is repeated the repetitions are simply deleted since the iteration controls will re-invoke it if necessary.

While [106] says that large SCCs are hierarchically decomposed to find smaller SCCs embedded in the larger ones, it does not describe precisely how to remove back edges to do this. The method used here is to remove the back edge with the largest difference in depth-first-search finishing order between its target vertex and its source vertex. The intuition is that larger differences are associated with larger cycles in the signal graph. With this choice of edges, the overall scheduling algorithm takes no more than $O(V_S(V_S + E_S))$ and at least $\Omega(V_S + E_S)$ time.

Figure 3.15 shows the generated code for scheduling and invocation operations. There is a statically-initialized scheduling information array which indicates both receiving codeblocks and indices of flags to use when a signal value changes. The flag indices are included because a two-dimensional array of flags indexed on codeblock and level would be highly sparse and cache-inefficient.

One would expect to see that levelized event-driven scheduling will reduce the number of codeblock invocations relative to dynamic scheduling. As was the case for acyclic scheduling, this expectation must be tempered by the fact that invocation coalescing may proceed differently than in pure dynamic scheduling. It is also possible that the increased complexity and overhead

▷ *SchedInfo* is a statically generated array of scheduling
information for each signal. Each element has fields:
codeblock: the receiving codeblock
flagIndex: index of flag variable to use
level: level at which to schedule
▷ *flags* is an array of booleans which keeps track of which
codeblock/level combinations have already been scheduled.
▷ *levels* is the schedule structure, which is an array of lists.
Each list element has a field *codeblock* indicating the
codeblock and an integer *flagIndex* indicating which flag
variable should be used
▷ *masterLevel* is a global variable indicating the current level being invoked
▷ *numLevels* is the number of levels in the schedule
▷ *runAgain* is a global boolean flag

LEVELIZED-SCHEDULE(S)
▷ S is the signal whose value has changed
flagIndex ← $SchedInfo[S].flagIndex$
if $flags[flagIndex] =$ FALSE
do $flags[flagIndex] \leftarrow$ TRUE
$lev \leftarrow SchedInfo[S].level$
ENQUEUE($levels[lev]$, $\{SchedInfo[S].codeblock, flagIndex\}$)
if $lev \leq masterLevel$
do $runAgain \leftarrow$ TRUE

LEVELIZED-INVOKE()
ENQUEUE-FORCED-INVOCATIONS() ▷ as per Section 3.5.4
repeat $runAgain \leftarrow$ FALSE
for $masterLevel \leftarrow 1$ **to** $numLevels$
do *foreach* C **in** $levels[masterLevel]$
do $flags[C.flagIndex] \leftarrow$ FALSE
Invoke $C.$ *codeblock*
until $runAgain =$ FALSE

Figure 3.15: Levelized Event-driven scheduling operations

of levelized scheduling will cause the improvements in invocations to not result in significant speedup because the decrease in codeblock invocations will be competing with an increase in the average cost of scheduling and dispatching a codeblock.

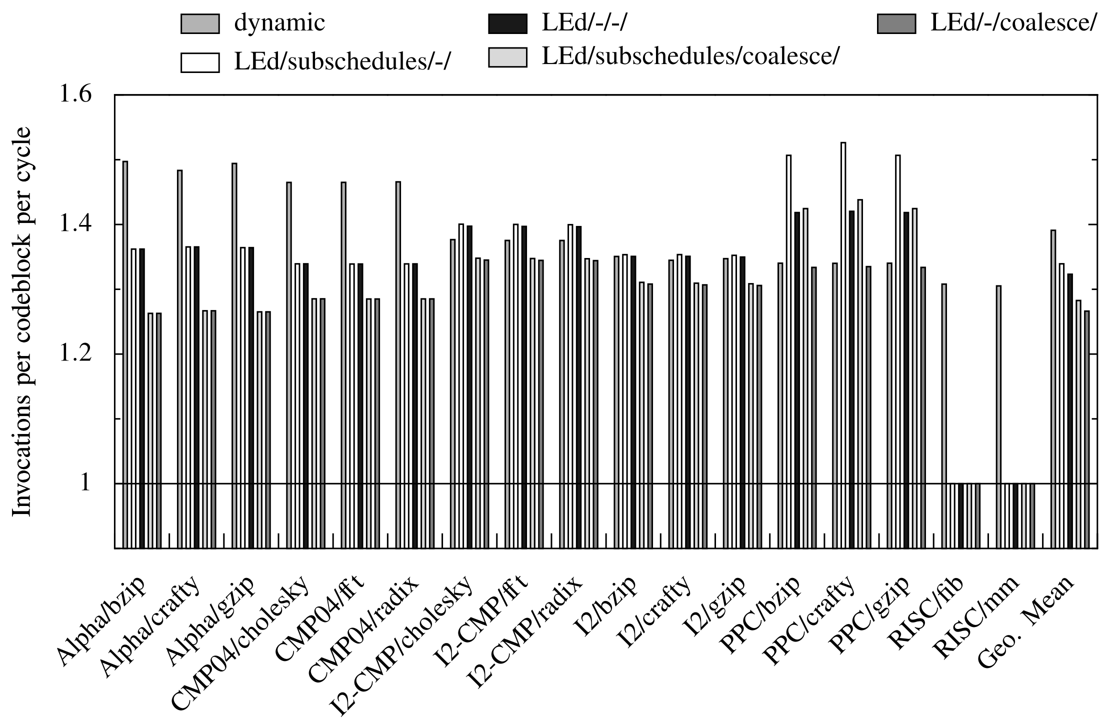
Figure 3.16(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination for dynamic scheduling and levelized event-driven scheduling performed both with and without embedded subschedules and with and without subgraph-based coalescing. Figure 3.16(b) shows the speedup of each of these techniques vs. dynamic scheduling. Averages are on the right-hand side of the graph and are geometric means across all models and benchmarks.

As was the case for acyclic scheduling, the results segment the models into two classes: those that have more codeblock invocations with levelized event-driven scheduling (***I2***, ***I2-CMP***, and ***PPC***) than with dynamic scheduling and those that have fewer invocations (***ALPHA***, ***CMP04***, ***RISC***). These classes also correspond to the models which have and do not have cycles in their signal graphs.

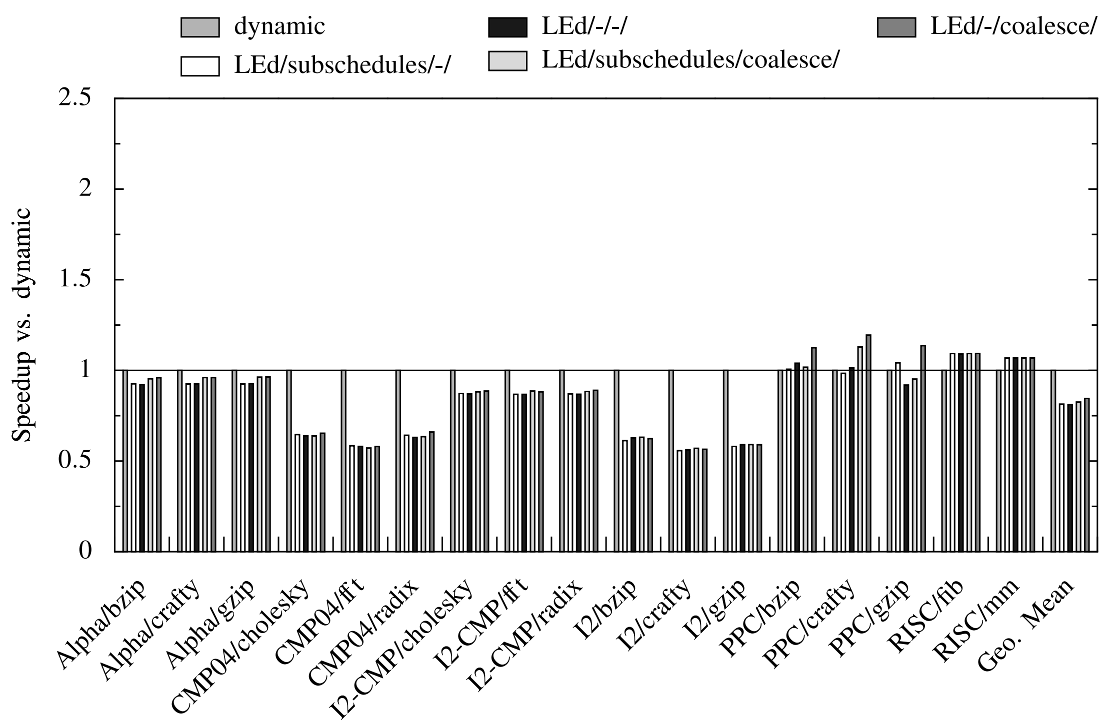
Levelized event-driven scheduling does succeed in reducing the number of codeblock invocations for the models with acyclic signal graphs. However, speedup is obtained within this class only for the ***RISC*** model, and that speedup is very low. Subgraph-based invocation coalescing can further reduce invocations, but the speedup is only marginally affected. Subschedule embedding is, of course, irrelevant when the signal graphs are acyclic.

The main factor causing this poor showing is the higher overhead inherent in the more complex data structure used for scheduling. This does not fully explain the pathological behavior of the ***CMP04*** model. Profiling shows that 3.5 times as many cache misses are being generated by the levelized event-driven simulator than the dynamically scheduled simulator. This is because the larger scheduling data structure which levelized event-driven scheduling requires places much more pressure on the caches. This could be also be considered overhead. Rerunning the simulator binaries on a different host system with a different cache hierarchy shows results similar to those of the ***ALPHA*** model.

When the signal graph has cycles, subschedule embedding increases the number of invocations. This is because the fixed, invoke-all-codeblocks behavior of subschedules invokes some



(a) Invocations per codeblock per cycle



(b) Speedup vs. dynamic

Figure 3.16: Levelized Event-driven (LEd) scheduling results

codeblocks in the cycles too many times. As when the signal graph was acyclic, subgraph-based coalescing does reduce the number of invocations; without it levelized event-driven scheduling results in more invocations than dynamic scheduling.

The impact upon speedup is mixed; for the ***I2*** and ***I2-CMP*** model, the use of subschedule embedding or coalescing causes little difference and no technique provides speedup. The ***I2*** model behaves similarly to the ***CMP04*** model; there levelized event-driven scheduling slows down simulation because of cache effects. On the other hand, the ***PPC*** model sees differences between techniques and faster simulation speeds even though the number of invocations may increase. Levelized event-driven scheduling has serendipitously reduced the number of invocations of particularly expensive-to-execute codeblocks while increasing invocations for the cheaper ones, resulting in a net performance improvement.

These results do not reflect those in [106] for logic simulation, where levelization both reduced the number of gate evaluations by one half to one third and was from 8 to 77 times faster than dynamic scheduling. The reasons for this are twofold. First, their experimental methodology allowed selective-trace execution across clock cycles (test vector applications in their case). Second, they generated specialized assembly code for the data structure manipulations and generate all code for gates within a single function, allowing very low-overhead scheduling and invocation.

3.6.4 Partitioned Scheduling

The partitioned scheduling algorithm described in [21] has been implemented with the addition of dynamic subschedule embedding as described in Section 3.5.2. Signal-based coalescing as in [21] is also used; however signals are considered in the the order in which they appear in the schedule, not numeric order. Figure 3.17 shows the generated code for scheduling and invocation operations; note that in many cases the code generator can specialize the scheduling code so that a lookup of the $doDynamic$ flag in the scheduling information is not needed. The dynamic invocation code is invoked for each embedded dynamic subschedule.

One should expect to see that partitioned scheduling reduces the number of codeblock invocations relative to dynamic scheduling. This expectation is qualified by the fact that invocation coalescing (which is implicit in dynamic scheduling) may proceed differently even for an acyclic

$\triangleright$ *SchedInfo* is a statically generated array of scheduling information for each signal. Each element has fields:
    codeblock: the receiving codeblock
    doDynamic: is dynamic scheduling required?
$\triangleright$ *StaticSched* is the static schedule; each element has fields:
  codeblock: the codeblock to invoke
  force: should invocation be forced?
$\triangleright$ *SchedList* is a FIFO queue.
$\triangleright$ *flags* is an array of codeblock flags with values
  TRUE, FALSE, and NEEDED

```
PARTITIONED-SCHEDULE(S)
  ▷ S is the signal whose value has changed
  if SchedInfo[S].doDynamic = TRUE
    then if flags[C] = FALSE
           then flags[C] ← TRUE
                ENQUEUE(SchedList, C)
  elseif selective-trace-enabled
    then flags[C] ← NEEDED
```

```
PARTITIONED-INVOKE()
  ▷ Run static schedule
  in-acyclic-finish ← FALSE
  foreach E in StaticSched
      do
         if not selective-trace-enabled
           then Invoke E.codeblock
         elseif E.force = TRUE or flags[E.codeblock] ≠ FALSE
           then flags[E.codeblock] ← FALSE
                Invoke C
```

```
DYNAMIC-INVOKE()
  ▷ This code is used for embedded dynamic sections
  ENQUEUE-FORCED-INVOCATIONS()   ▷ as per Section 3.5.4
  while SchedList not EMPTY
      do C ← DEQUEUE(SchedList)
         flags[C] ← FALSE
         Invoke C
```

Figure 3.17: Partitioned scheduling operations

signal graph. The reduction in invocations should translate to improved simulation speed through two factors: the reduction in the number of invocations and a large reduction in the average cost of scheduling a codeblock.

Figure 3.18(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination for dynamic scheduling and partitioned scheduling with and without selective-trace. Figure 3.18(b) shows the speedup of each of these techniques vs. dynamic scheduling. Averages are on the right-hand side of the graph and are geometric means across all models and benchmarks. Note that no dynamic schedules are required for any of these models.

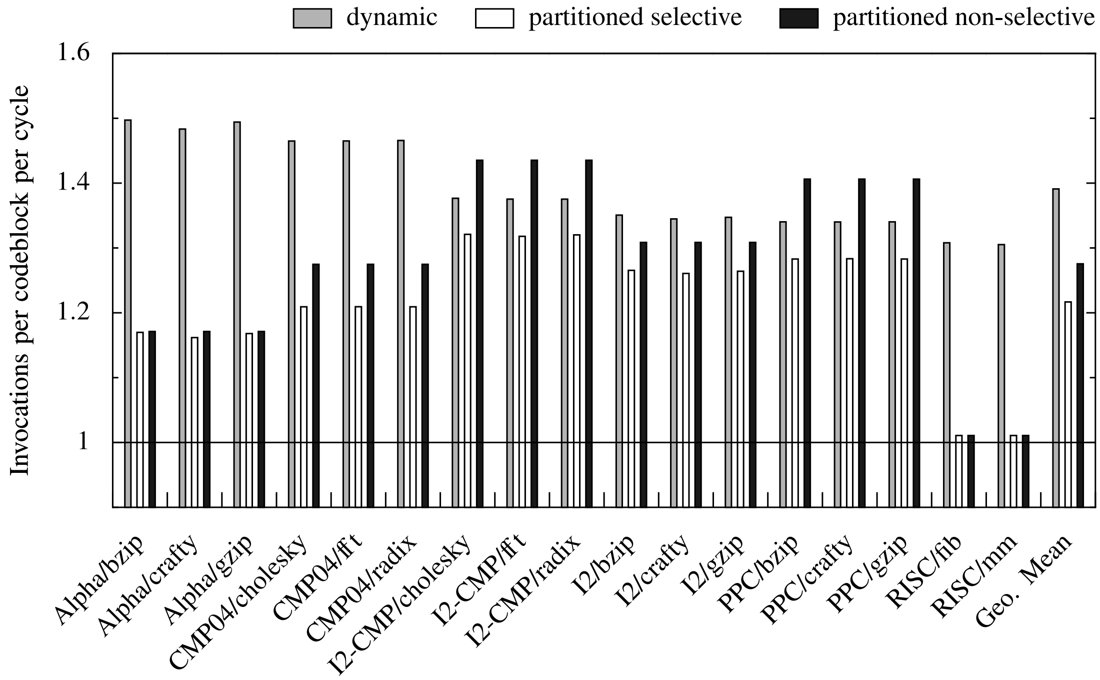
These results divide the models into two classes: those which have a larger number of invocations when non-selective partitioned scheduling is used (***I2-CMP*** and ***PPC***) and those which have fewer.

Partitioned scheduling is quite effective at reducing the number of codeblock invocations for the larger class. The number of codeblock invocations drops without the use of selective-trace techniques by up to 24%. This results in speedups from 1.22 to 2.08. The speedup is generally larger than the relative decrease in codeblock invocations, indicating that overhead has been reduced as well. For these fully static schedules, the overhead with non-selective-trace partitioned scheduling should be zero.

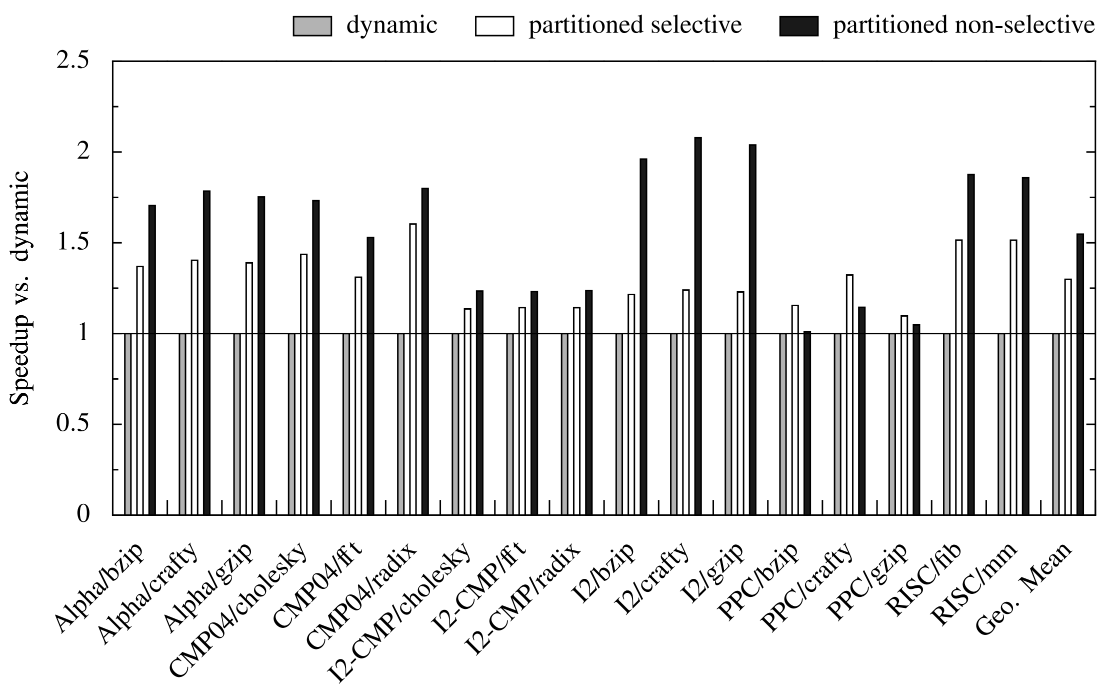
As with acyclic scheduling, selective-trace reduces the number of codeblock invocations relative to non-selective-trace, but the reduction in invocations is usually outweighed by the increase in overhead. Note that this is true even though the overhead is simply the setting and testing of a flag.

The ***RISC*** model shows an interesting behavior; the number of codeblock invocations is not reduced to precisely one invocation per codeblock per cycle with partitioned scheduling. A single codeblock is called twice. The reason for this is that the situation is like that of Figure 3.9; signal-based coalescing was unable to coalesce these two invocations because another invocation upon which the second depends could not be coalesced.

The other class of models (***I2-CMP*** and ***PPC***) has much lower speedup than other models as well as an increase in the number of codeblock invocations relative to dynamic scheduling when



(a) Invocations per codeblock per cycle



(b) Speedup vs. dynamic

Figure 3.18: Partitioned scheduling results

non-selective-trace is used. In both models, the signal graph contains small cycles; in the case of ***PPC***, there are 26 of them. The increase in codeblock invocations occurs because the partitioned scheduling algorithm "over-schedules" these blocks relative to their true data dependences, just as in the example given in Section 3.5.2. Also, as the number of codeblocks involved is actually a small fraction of all the codeblocks to be scheduled but these codeblocks have had a large impact on invocations, the disruptive impact of cycles on invocation coalescing may be high. Selective-trace does reduce the invocations to below the level in dynamic scheduling, though only for the ***PPC*** model does this result in speed improvements.

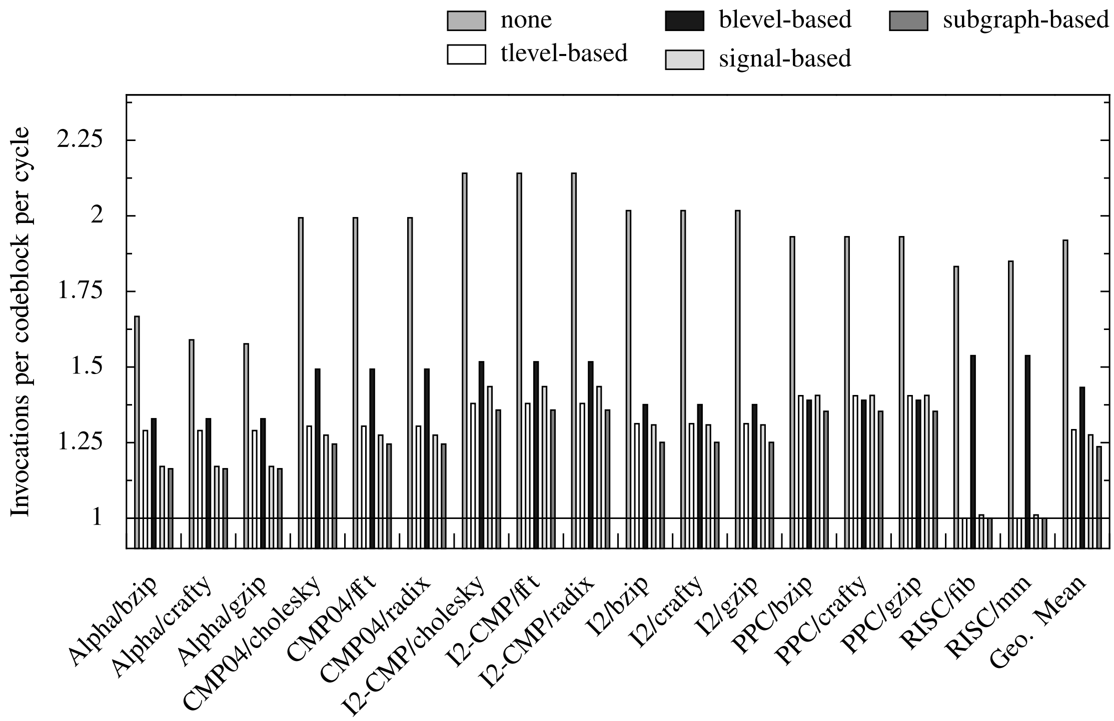
Improving invocation coalescing

As discussed in Section 3.5.3, it is possible to improve scheduling results further through improved invocation coalescing. Four coalescing strategies are evaluated: tlevel-based as used in acyclic scheduling, a very similar blevel-based technique, signal-based as proposed by Edwards, and the novel subgraph-based technique proposed in Section 3.5.3.

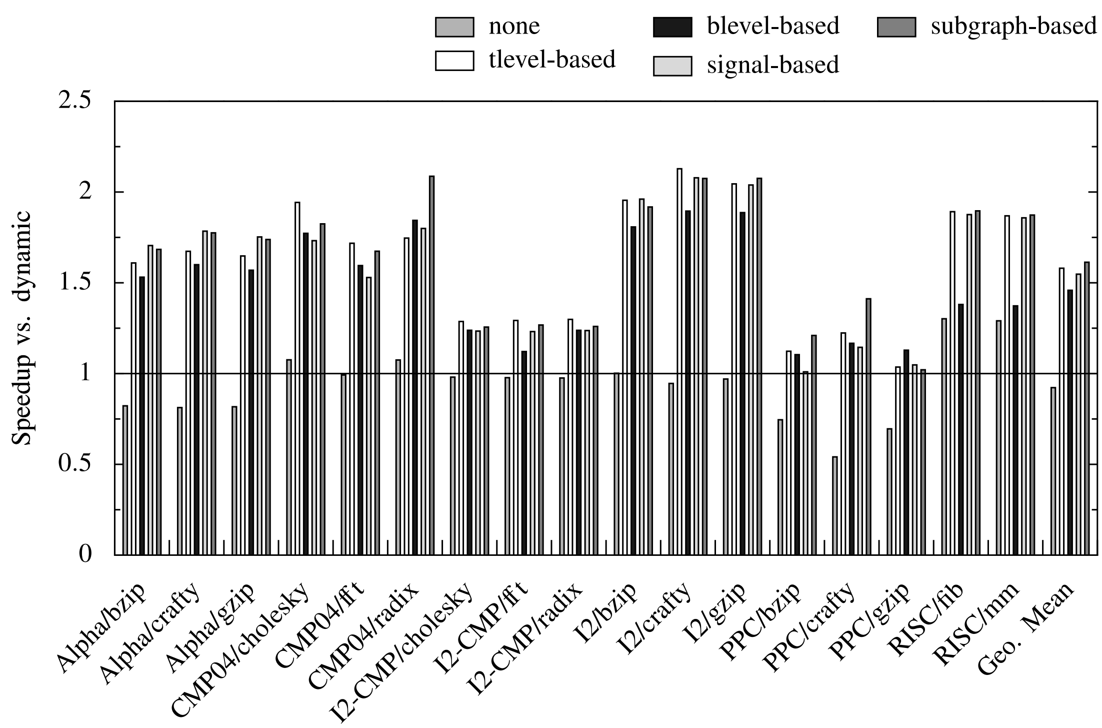
Figure 3.19(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination for partitioned scheduling with the four different coalescing techniques as well as no coalescing.. Figure 3.19(b) shows the speedup of each of these techniques vs. dynamic scheduling. Averages are on the right-hand side of the graphs and are geometric means across all models and benchmarks.

Coalescing is very important for all the models and *always* produces better results for these models. The speedup over not coalescing ranges from 1.15 to 2.61 with an average of 1.68. For several of of the models, not performing coalescing even causes the statically scheduled simulator to be slower than the dynamically scheduled simulator, thus sabotaging the static scheduling process.

The different coalescing techniques produce speed and invocation differences. Subgraph-based coalescing always results in the fewest invocations because of the additional flexibility provided by the movement of subgraphs. However the speed difference versus other techniques is modest: only a 4.5% average speedup versus signal-based or 1.9% speedup versus tlevel-based coalescing. Blevel-based coalescing usually under-performs all the other techniques. Signal-



(a) Invocations per codeblock per cycle



(b) Speedup vs. dynamic

Figure 3.19: Invocation coalescing results

based coalescing usually results in fewer invocations than tlevel-based coalescing, but neither tlevel-based nor signal-based coalescing are clearly superior to each other in speed.

While a large reduction in invocations leads to improved speed, small reductions do not always do so. For example, subgraph-based coalescing yields a slower simulator for the ***CMP04*** model than tlevel-based coalescing even though it yields fewer invocations. This is because different techniques coalesce different invocations, but not all coalescings are of equal worth. Coalescing costly invocations leads to more speedup than coalescing cheap invocations. None of the techniques consider heterogeneity of invocation cost.

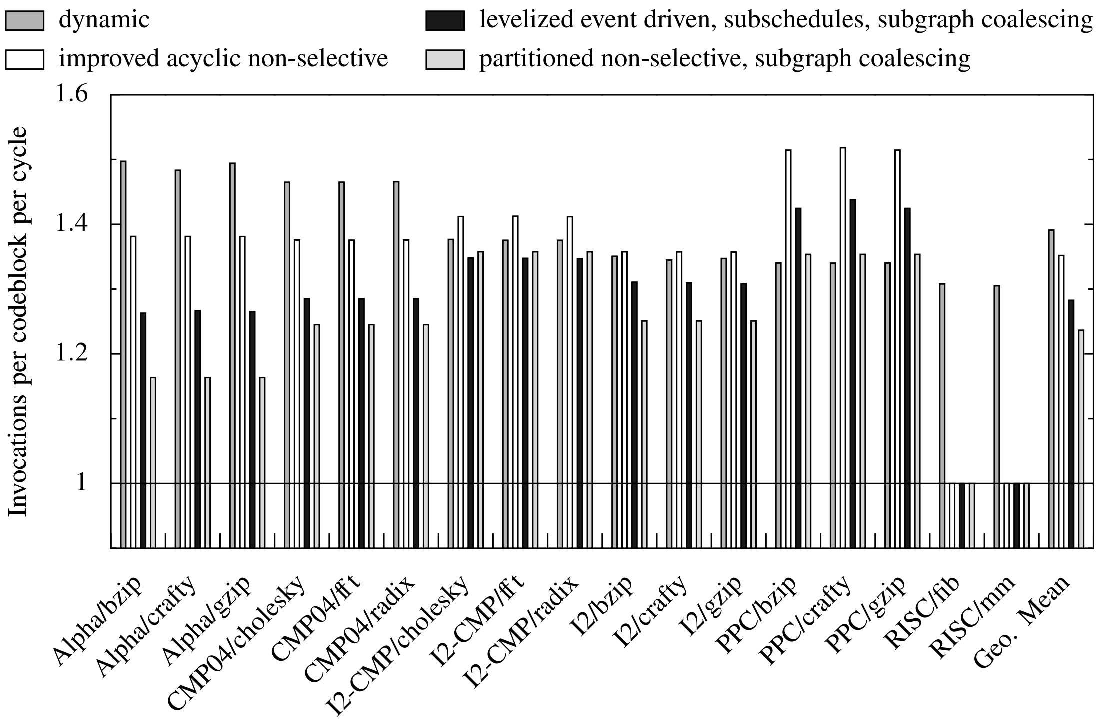
To summarize, performing some form of invocation coalescing is very important, and blevel-based coalescing should not be used. Subgraph-based coalescing has the best overall performance, but tlevel-based coalescing is a lower time-complexity algorithm with little difference in performance.

3.6.5 Comparing the Techniques

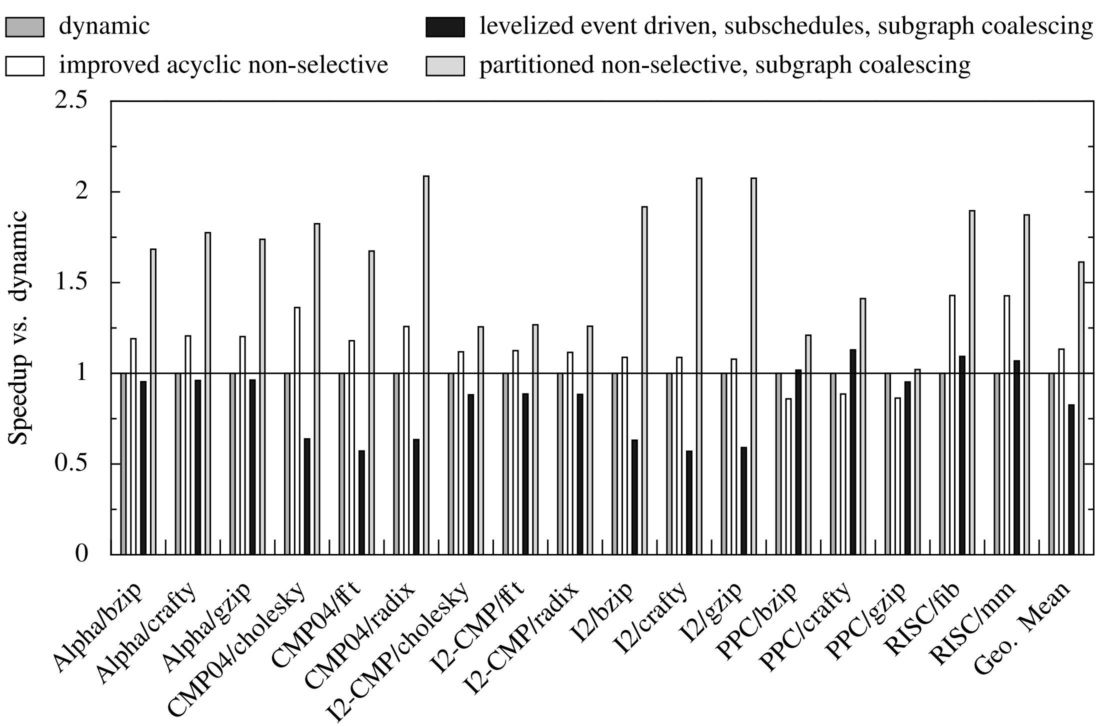
Figure 3.20(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination for the best overall configuration for each type of scheduling: dynamic, improved acyclic without selective-trace, levelized event-driven with subschedules, and partitioned without selective-trace with subgraph-based invocation coalescing.. Figure 3.20(b) shows the speedup of each scheduling technique vs. dynamic scheduling. Averages are on the right-hand side of the graph and are geometric means across all models and benchmarks.

The ***RISC*** model highlights the differences in scheduling and invocation overhead between the techniques. All of the technique are able to reduce the ***RISC*** model to a single invocation per codeblock. However, the speedup results vs. dynamic scheduling are dramatically different. Levelized event-driven scheduling has only a tiny 1.04 speedup, acyclic scheduling has a respectable 1.43 speedup, but partitioned scheduling has a very high 1.89 speedup. These differences stem directly from the differences in overhead for the different scheduling techniques.

For the other models, partitioned scheduling has a double advantage: it both produces a lower number of codeblock invocations than the other static or hybrid techniques and has lower



(a) Invocations per codeblock per cycle



(b) Speedup vs. event-driven

Figure 3.20: Overall technique comparison

overhead, allowing it to obtain much higher speedups in nearly all cases. When there are many cycles in the signal graph, such as in the ***PPC*** model, partitioned scheduling is the only technique able to consistently obtain speedup.

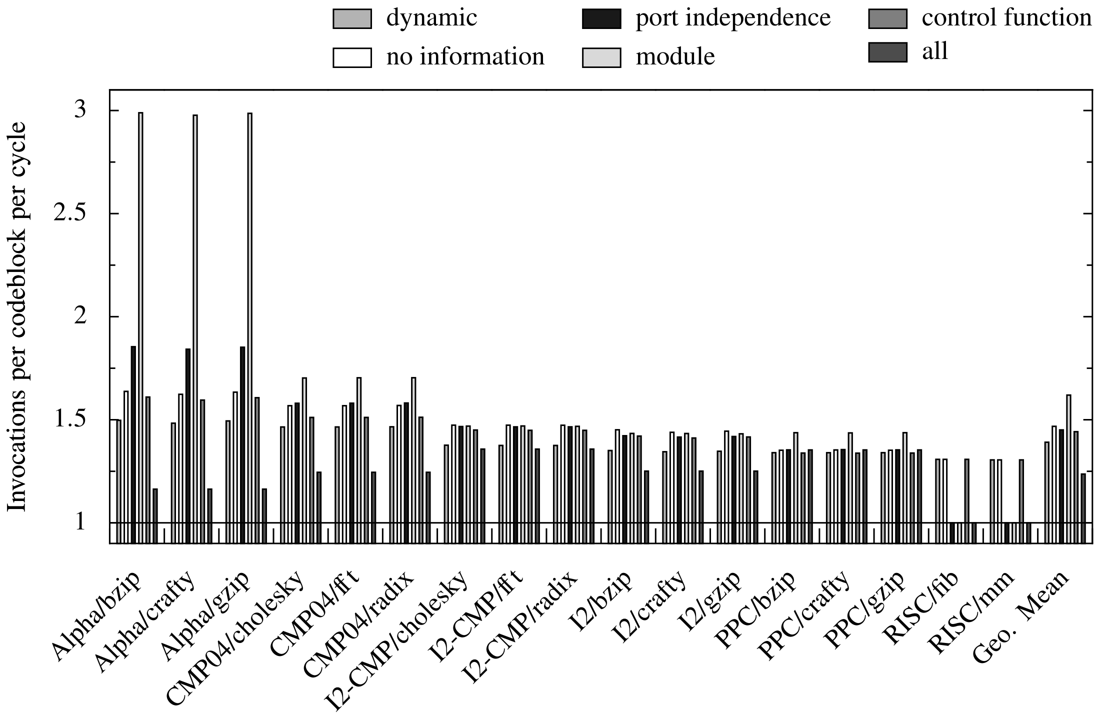
The large overhead required to manipulate data structures in the levelized event-driven technique puts it at a distinct disadvantage with respect to both acyclic and partitioned scheduling. It is only able to obtain speedup for the single model with an acyclic codeblock graph and thus is not suitable for microarchitectural simulation.

3.6.6 The Importance of Dependence Information Enhancement

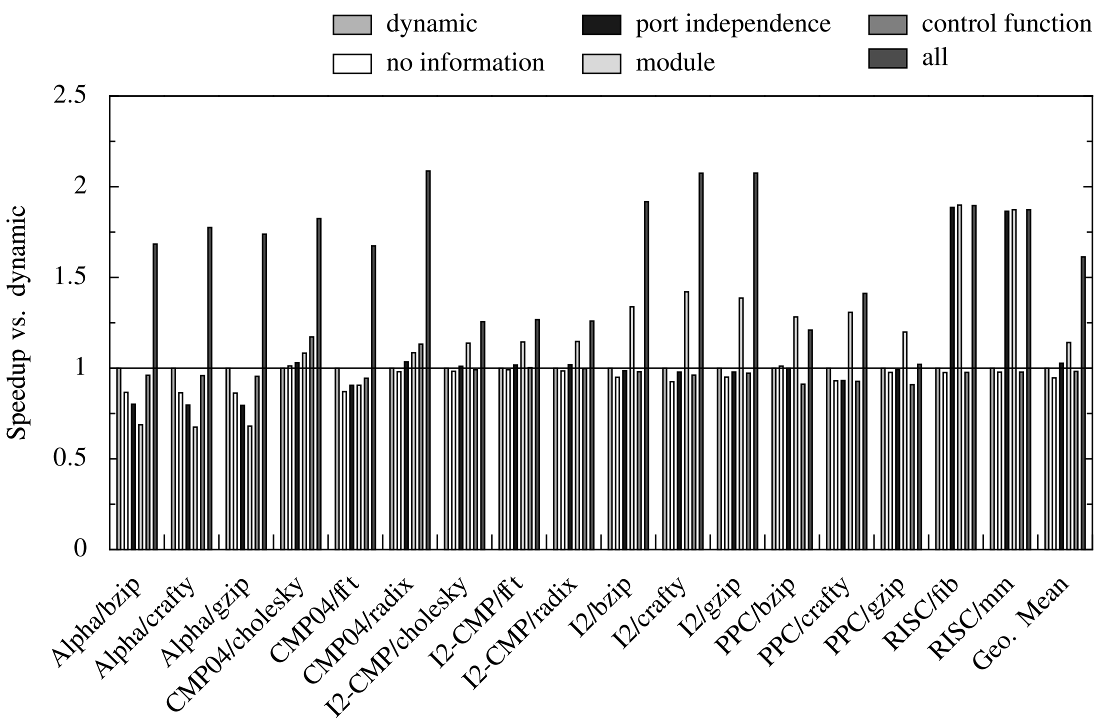
All of the scheduling techniques rely upon knowledge of computational dependences between signals. The results heretofore shown used all the dependence information enhancement techniques proposed in Section 3.5.1. This section evaluates each of those techniques when used with partitioned scheduling. The levels of enhancement which are compared are none, only port independence annotation, only module annotations, only control function analysis, and all enhancements. When some enhancements are not used, embedded dynamic subschedules may be included in the schedule.

Figure 3.21(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination with the different enhancements: none, port, module, control function, and all. Figure 3.21(b) shows the speedup of each scheduling technique vs. dynamic scheduling. Averages are on the right-hand side of the graph and are geometric means across all models and benchmarks.

Partitioned scheduling without any dependence information enhancement is essentially dynamic scheduling because in the absence of information to the contrary, the default flow-control signals between any connected module instances lead to cycles in the signal graph. The result is that nearly all signals get pulled into a handful of large strongly-connected components, and often just one such component. That any signals at all can be statically scheduled is due to the presence of unconnected port instances whose signals are treated as constant values and thus provide a very small amount of dependence information. Note that these results are not the same as



(a) Invocations per codeblock per cycle



(b) Speedup vs. dynamic

Figure 3.21: Dependence information enhancement

those previously presented for dynamic scheduling because those results included use of all the dependence information enhancements; this allowed some forced invocations to be dropped.

Use of all of the dependence information enhancements decreases the number of codeblock invocations and improves speed by up to 119% over having no enhancements. Having the additional information results in a fully static schedule for all of these models. The additional information can also show that some signals are not used and drop them from the schedule. The improvement in speed is generally larger than the relative decrease in invocations; this is because a fully static schedule has lower overhead for scheduling than a partially dynamic schedule.

Partial information enhancement presents a more complex picture. The behavior is tied to the size of strongly-connected components which are created in the signal graph when dependence information is missing. This difference in component size stems from differences in the way in which portions of the model which are missing information are connected and leads to differences in the proportion of invocations taking place in dynamic subschedules. Table 3.6 shows the proportion of signal evaluations taking place in dynamic subschedules for each of the models for partial information enhancement.

	Dynamic evaluations		
Model	Port	Module	Control Function
ALPHA	97%	26%	88%
CMP04	99%	73%	96%
I2	88%	27%	95%
I2-CMP	90%	42%	97%
PPC	91%	30%	100%
RISC	0%	0%	100%

Table 3.6: Proportion of signal evaluations scheduled dynamically

When only control function analysis is used, the computational dependences within the module instances are not known. Module instances tend to have many signals and are usually connected directly to one another. As a result, the signal graph comes to have very large strongly-connected components within it. These components cause large embedded dynamic subschedules to be generated, which behave very similarly to having no information at all, as indicated by the high proportion of dynamic invocations. The number of invocations is similar, though there is some speedup for the ***ALPHA*** and ***CMP04*** models, which have slightly larger decreases in invocations.

When only module annotations are used, the computational dependences within the control functions are not known. Control functions have few signals, are much less common than module instances, and are rarely connected directly to one another. The resulting signal graph has small strongly-connected components which are statically scheduled. This leads to a lower proportion of dynamic invocations. However, because information was missing, the static schedules for these small cycles are not optimal and cause there to be too many invocations. Recall that this problem appeared in the ***PPC*** model for full information due to a few control functions that could not be analyzed. Here all control functions are involved. As a result, for all the models except the ***RISC*** model the number of invocations is more than when only control dependences are used. For most models the number of invocations is not greatly affected relative to no information, but for the ***CMP04*** and ***PPC*** models the number of invocations does increase somewhat, indicating that the schedules are poorer. Nevertheless, module annotations nearly always provide speedup over no information because the significant amount of static scheduling improves the average cost of scheduling. The exception is the ***ALPHA*** model, which will be discussed later.

When the scheduler uses only port independence annotations, very little is known about computational dependences within the modules and nothing at all about the control functions. The result is a schedule with a very high dynamic proportion of invocations and correspondingly close behavior to no information at all. The only anomaly is the ***ALPHA*** model, which has more invocations and less speed when port independence is used. This happens because the independence information is enough to split the signal graph into two strongly-connected components, but many signals which are scheduled to do be generated during the second subschedule are actually generated in the first embedded subschedule. As a result, all but one of the forced invocations for the second subschedule are extraneous.

The ***RISC*** model appears odd in that port dependence or instance annotations give the same results as all annotations and control points analysis gives the same result as no information. This is because the ***RISC*** model has no control functions, never uses the ack signals, and all output signals of W-codeblocks depend upon all their used input signals; thus all computational dependencies can be inferred from port independence or instance annotations alone.

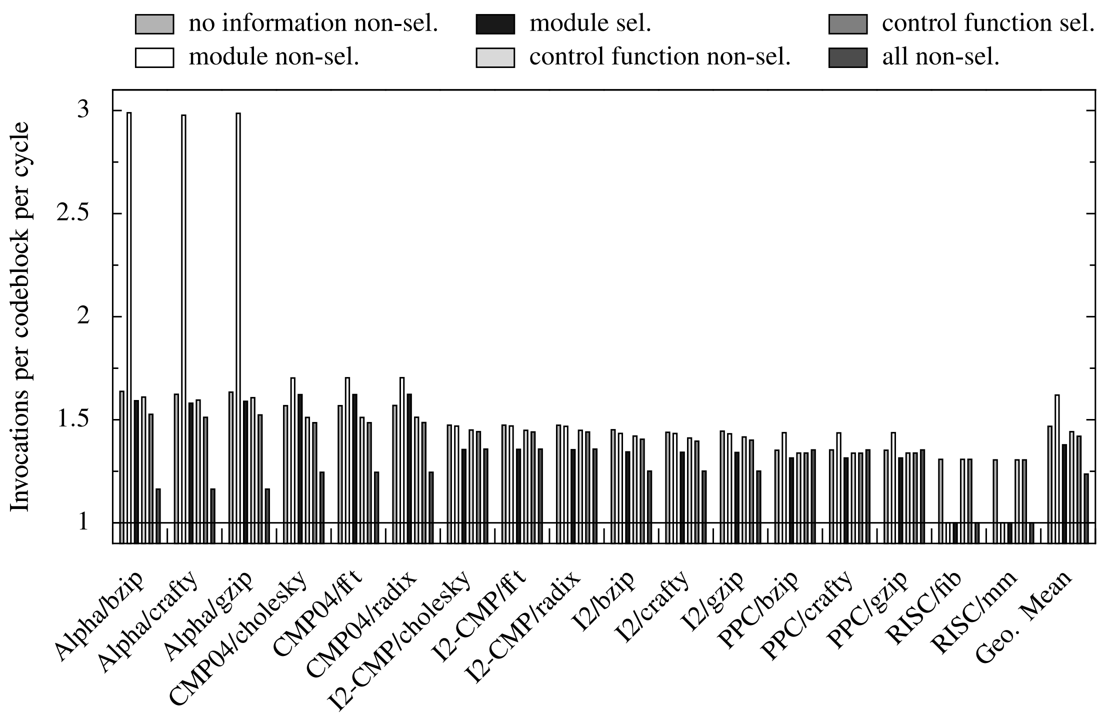
The strangest results are those for the ***ALPHA*** model. When only module annotations are used the number of invocations climbs dramatically and the speed of the simulator falls. The resulting signal graph has a strongly-connected component with 157 signals. This number of signals is very close to the size of a large component which would trigger an embedded dynamic subschedule. The hierarchical partitioning algorithm chooses a partitioning which is nested three-deep but never has more than two signals in the head. As a result, no portion of this component is converted to a dynamic section and codeblocks at the deepest nesting level are scheduled to run up to 11 times. These results suggest strongly that the either the definition of large components is set too high and/or that the level of nesting ought to be considered when deciding whether to convert to dynamic sections while unrolling the schedule.

Using selective-trace to deal with small cycles

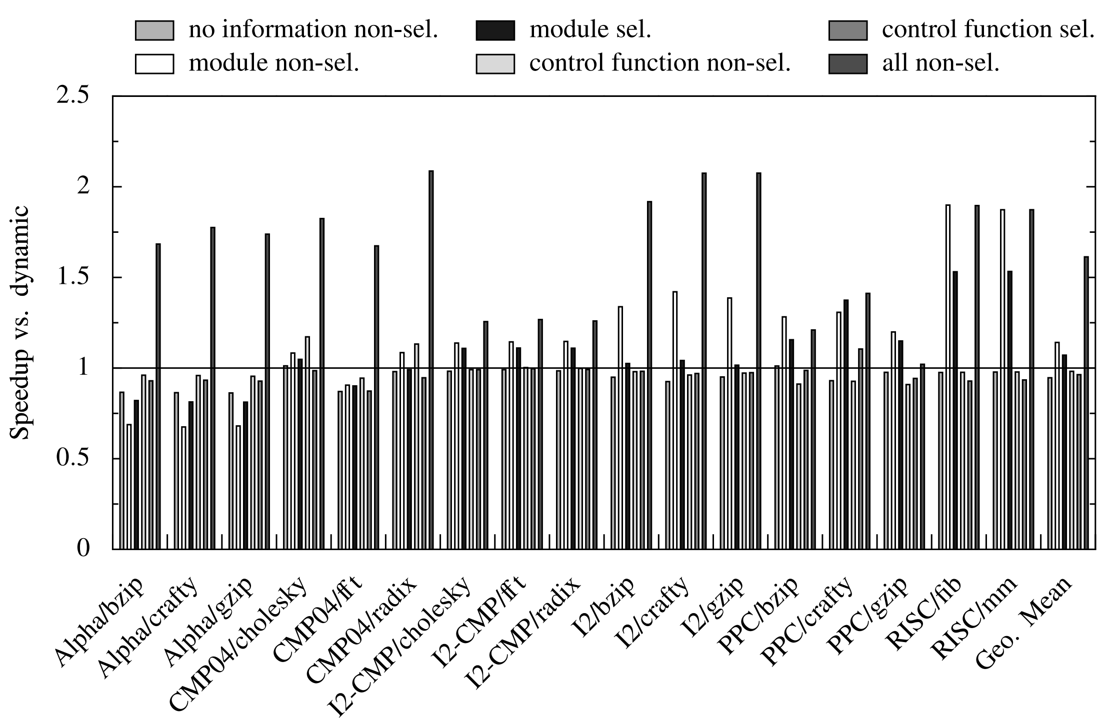
Not having full dependence information may be a common situation: users may not wish to devote time to annotating their custom modules or may not write control functions in an easily parseable way. Indeed, the effects of this appear in the ***PPC*** model, where unparseable control functions tend to cause the small cycles present in the signal graph. As a result, it is desirable to see whether the extra invocations which result from small and medium-sized cycles can be reduced using selective-trace techniques.

Figure 3.22(a) shows the average number of codeblock executions per codeblock per simulated cycle for each model and benchmark combination for module annotations and control function analysis with and without selective trace, and no enhancements and all dependence information enhancements without selective trace. Figure 3.22(b) shows the speedup of each scheduling technique vs. dynamic scheduling. Averages are on the right-hand side of the graph and are geometric means across all models and benchmarks.

Selective trace reduces the excess codeblock invocations caused by poor static schedules for cycles effectively, but this does not translate into performance gains unless there were many extra invocations, such as happens for the ***ALPHA*** model when module annotations are used. This is because of the increase in scheduling and invocation overhead when selective trace is used.



(a) Invocations per codeblock per cycle



(b) Speedup vs. no enhancement

Figure 3.22: Dependence enhancement with selective trace

3.7 Summary: Scheduling for Uniprocessor Structural Simulation

In this chapter I have shown that static and hybrid uniprocessor scheduling techniques can be used successfully to increase structural simulation speed for the Heterogeneous Synchronous Reactive model of computation. Both acyclic scheduling and partitioned scheduling are effective at both reducing the number of codeblock invocations and increasing simulator speed, providing speedups of up to 1.44 and 2.09 respectively over dynamic scheduling. Levelized event-driven scheduling does not reliably improve performance, though it does decrease codeblock invocations. Selective-trace techniques reduce the number of codeblock invocations, but generally do not provide improved performance because of the increased overhead they imply.

Contrary to prior belief, partitioned scheduling can generate correct static schedules for the most common model of computation, the zero-cycle Discrete Event MoC, when models are restricted to be microarchitecturally synchronous. With modifications, partitioned scheduling can generate correct static schedules for the larger class of models which are logically synchronous if the signals can be represented in a fixed number of bits. Thus all microarchitectural models which describe synchronous designs where signals can be represented in a fixed number of bits can be statically scheduled using partitioned scheduling

Four enhancements to scheduling techniques are needed to make scheduling practical: dependence information enhancement to improve the precision of signal graphs used for scheduling, dynamic subschedule embedding to control scheduler execution time in the face of limited dependence information, subgraph-based invocation coalescing to reduce redundant work, and forced invocation to allow dynamic scheduling within an HSR framework.

Chapter 4

Scheduling for Parallel Structural Simulation

The previous chapter has shown how to increase the speed of structural microarchitectural simulators by scheduling concurrent codeblock invocations so as to reduce the number of invocations required. Further speed improvements may be obtained by scheduling these invocations to execute concurrently on parallel hardware. The structural and concurrent nature of structural simulation frameworks allows this to be done automatically. This chapter presents efficient scheduling of concurrency for a parallel structural microarchitectural simulator.

This chapter begins with an explanation of the importance of parallelization and related work on parallel simulation. It then describe what steps are required to carry out parallelization, and my approach to parallelization of simulators generated using the Liberty Simulation Environment, showing how the structure of the model yields information which allows automatic parallelization. Next, it classifies the problem of statically scheduling simulation tasks as a variant of multiprocessor task scheduling and discuss related work from that domain. Finally, it introduced new heuristics for solving the multiprocessor task scheduling problem with precedence constraints, resource constraints, and sequence dependence and evaluate these heuristics on parallel LSE simulators running on a traditional multiprocessor system, a chip multiprocessor, and a simultaneous multithreading processor.

4.1 Motivation: Why Parallel Simulation?

Before parallelizing any application, one should determine whether parallelization is worthwhile. Microarchitectural simulation is commonly used in situations where there are fewer available processors than jobs to be run, thus throughput is the main consideration. In such situations, job-level parallelization (running independent simulations in parallel) is generally thought to offer more throughput than parallelization of the simulator itself, as parallelization of a single application is rarely perfectly efficient. However, this argument fails to take into account three valid reasons for parallelizing a simulator:

- At times, latency matters more than throughput. For example, when a new microarchitectural technique is being evaluated, a common methodology is to run a small number of simulations, make a design change, and then simulate again. In such situations, any reduction in latency improves productivity.

- Parallelization may increase the problem size that can be effectively tackled. Scientific codes have sometimes used message-passing-based parallelization to work on data sets larger than physical memory on a single machine. While this dissertation does not investigate a message-passing model of parallelization, the principle also holds for caches in shared-memory multiprocessors: parallelization can increase the effective cache size or decrease the per-processor working set size, and as shown later in this chapter, can lead to superlinear speedups.

- The assumption that independent jobs have independent performance may not be valid when hardware resources are shared. Current trends in processor design emphasize multithreading cores and multicore processors. These cores typically share some resources such as execution units or caches. When they do, independent jobs may not run more efficiently than a parallelized job; indeed, situations where executing independent jobs on a multicore processor lead to profound effects on each job's execution time have been demonstrated[13].

Because the systems which microarchitects build are themselves parallel, assuming that simulators of these systems will exhibit parallelism is reasonable. However, extracting and exploiting this parallelism presents challenges. Parallelization is a non-trivial task, usually requiring much manual programmer effort. Thus automatic parallelization of the simulator is a worthy goal. This goal has been pursued for decades, but success has been elusive except for highly-regular and analyzable codes[108]. Continuing in this tradition of specialized solutions for a given class of codes, I will show that the structure inherent in a structural model provides a rich source of information about the parallelism available in the simulator which can be exploited to automatically generate a parallel simulator and obtain improvements in simulator speed.

4.2 Related Work: Parallel Simulation

The related work on parallel simulation falls into three categories: distributed discrete-event simulation, parallel microarchitectural simulation, parallel sampled simulation, and parallel structural simulation frameworks. This section describes each of these fields.

4.2.1 Distributed Discrete-Event Simulation

The phrase "parallel simulation" commonly refers to a distributed discrete event simulation, or techniques for parallelizing simulators which use the Discrete Event model of computation. There is an extensive literature on parallel and distributed discrete-event simulation, only some of which can be discussed here. Overviews can be found in [30] and [27].

Distributed DE simulators typically execute in a very dynamic fashion, with invocations of codeblocks being driven by events. Different processors may be executing different timesteps; good performance depends upon concurrent execution of events occurring at different simulated times. The main problems to be solved are how to coordinate concurrent execution so that causality is not violated and how to properly load-balance the work across processors when the activity factors of the events are not known and change dynamically.

The simplest distributed simulation algorithms are **synchronous**. They use global event queues and a single global clock. Individual processors remove and insert tasks from the global queue[87]. This approach provides implicit load balancing and time agreement, but serializes execution at the event queue manipulations, generally leading to poor performance. Work stealing algorithms[90] have been proposed which maintain per-processor event queues, but global clocks are still required.

Asynchronous algorithms break this serial bottleneck by maintaining separate event queues and clocks in each processor. This implies, however, that mechanisms are needed to perform load balancing and synchronization of the clocks. Load balancing is carried out by assigning codeblocks statically to specific processors; how this is done is rarely described outside of specific domains such as logic simulation. The focus of most research has been better ways to perform clock synchronization.

Clock synchronization

The first asynchronous algorithms were known as **conservative** methods; the most well-known method is that of Chandy and Misra[14] and Bryant[11]. Codeblocks are assigned to processors by an unspecified partitioning algorithm. Each codeblock maintains its own clock.

Codeblocks send messages tagged with time values to each other. Messages are unbuffered and are thus sent via rendezvous synchronization: the sender and the receiver must both be waiting to perform the communication for it to occur. A codeblock waits to send an output message if it has one to send; it waits to receive an input message if the last time value received on that input is equal to the current local clock. The local clock is always the minimum across all signals of the last time value sent or received.

Deadlock situations can arise due to reconvergent fanout. To prevent these deadlocks, codeblocks must send **null messages** whenever they can deduce that they will send no output message for some amount of time. Chandy and Misra[15] later describe a deadlock handling mechanism which does not require null messages. The simulation is run until deadlock occurs and then a recovery phase is initiated to update local clocks.

Optimistic asynchronous algorithms were introduced by Jefferson[45]. His method, called **virtual time**, introduces input and output buffers and allows local clocks for each codeblock to speculatively run ahead of the messages which have been received. If a message is received which has a timestamp before the local clock, the codeblock must recover from its misspeculation by rolling back its state and sending **anti-messages** to cancel the messages which it has already sent. On receipt of an anti-message, a codeblock may need to perform rollback and send more anti-messages.

No global detection mechanism or recovery phase is required, but there is a need to purge old rollback state which is no longer required. A concurrent process is used to compute a conservative global virtual time which indicates what state can be purged. This same mechanism can be used for safely performing I/O or checkpointing system state. Note that rollback may require significant user effort to implement if large-granularity codeblocks are used. For this reason, optimistic techniques seem particularly ill-suited to microarchitectural simulation.

Effectiveness of parallel logic simulation

Soulé and Blank[90] look at several different parallel Discrete Event algorithms for logic simulation on shared memory hosts. They report that the synchronous algorithm performs particularly poorly due to overhead in accessing the event queue, giving a speedup of only about 2 for eight processors. A distributed event queue with work stealing performs better, yielding speedup of up to 9 for 15 processors. Parallelization of statically scheduled simulation gives linear speedup for some circuits, but not for others; strangely, they argue that because the circuits for which static parallelization performed the best were those for which it should have performed the worst due to activity factors, the technique should not be used. How partitioning is done is not described. Finally, they attempt a pipeline parallelization which works only if the circuit is acyclic and multiple input vectors are to be simulated, but which gives speedup of up to 10.5 for 15 processors.

Soulé and Gupta[91] measured the performance impact of the Chandy-Misra-Bryant algorithm for distributed logic simulation. They found that while there is much parallelism available to the simulator in logic circuits (from 42 to 200 gates could fire in parallel on average for their benchmark circuits) and the activity factors are quite low (1% to 3%), the Chandy-Misra-Bryant algorithm does not provide a great deal of speedup compared to uniprocessor simulation: less than 2 for eight processors. Their explanation for this poor result is that the number of gate evaluations increases, deadlock resolution takes a significant amount of time, and there is greater overhead in the distributed algorithm due to complex time checking on logic gate evaluation.

Partitioning for parallel logic simulation

Much research has been done on partitioning for parallel logic simulation. While the overall goal of partitioning has been to perform load-balancing, different techniques have tried to achieve this using a wide variety of other goals. These goals have included improvement in predicted load balance, improvement in concurrency, reduction of communication, formation of pipelines, reduction in null messages for conservative algorithms, and reduction in rollbacks for optimistic algorithms. Many different partitioning techniques have been reported; a sample of such techniques are here presented in approximately chronological order:

string partitioning[62, 52] Beginning at an input, find a string of connected gates which reach a primary output. Gates in a string must have exactly one fanin and one fanout in the string. Map the strings to processors randomly, then assign unassigned gates to strings so as to equalize the number of gates in a processor, trying to ensure that no more than one fanout of a particular gate is in any processor. The goal is to maximize concurrency in a synchronous distributed simulator by spreading fanouts across processors.

string partitioning with gate delays[3] Form linear strings as in string partitioning and assign them in round-robin fashion to processors. Assign remaining gates to processors such that gates assigned to the same processor have different delays from sibling gates at fanout points. Finally, move strings between processors until the difference in the number of gates assigned to each processor is lower than some tolerance in order to load balance. The goal is to reduce excess communication caused during string partitioning.

levelized[88] Calculate the *tlevel*[1] and assign gates with the same *tlevel* to the same processor. If there are more levels than processors, assign contiguous levels to the same processor. This creates a pipelined partitioning.

depth-first/breadth-first[73, 48] Perform a depth-first or breadth-first search on the gate-level netlist starting at either input or output gates and assign contiguous gates in the order in which they were encountered in the search to a processor. Also known as input/output partitioning. The goal is to reduce communication.

[1] As defined in Section 3.2: the longest path from a primary input to the gate

greedy levelized[73] Calculate the *tlevel* of each gate. In nondecreasing *tlevel* order, assign gates to the processor which minimizes a cost function which takes into account computation time and communication costs. As cost can be calculated incrementally, the algorithm is quite efficient. This is much like list scheduling, which will be discussed in much more detail in Section 4.4.2. The goal is to reduce communication for non-selective-trace compiled logic simulation.

annealing[73, 48] Form an initial partition using any technique you like. Then try to swap gates between processors, accepting swaps which cause a reduction in a cost function as well as some which increase cost with ever-decreasing probability. Different authors use different cost functions which may take into account computation time, communication costs, and null messages. The goal is set by the cost function [73] reports that annealing is not as effective as greedy levelized partitioning for non-selective-trace compiled logic simulation.

corolla[92] Identify and combine portions of the netlist where there is reconvergent fanout into non-overlapped subgraphs, called **corollas**. Attach gates which cannot be combined to the corolla which is the shortest path distance away. Assign corollas onto processors with an iterative clustering technique based upon the number of choices available for a given corolla to cluster; for example, if a corolla connects to one other corolla, cluster these two preferentially. The goal is to reduce rollbacks in optimistic algorithms.

concurrency-preserving[50] Break the netlist into disjoint subgraphs based upon a depth-first search from the input signals. Number each gate in such a way that all gates in a subgraph have consecutive numbers and heavily weighted edges between subgraphs have close numbers; a heuristic for the traveling salesman problem is used to do this. Assign gates with consecutive numbers into the same processor in such a way as to balance the total load. The goal is to reduce rollbacks in optimistic algorithms by load-balancing instantaneous workload.

multilevel[94] First, iteratively cluster connected gates in a step called coarsening. This attempts to reduce communication costs. Second, partition the clusters among processors taking into account only load balance. Third, break apart clusters iteratively and move individual gates or groups of gates across processors if this improves load balance and communication. The goal is to balance load, maximize concurrency, and reduce communication for optimistic algorithms.

4.2.2 Parallel Microarchitectural Simulation

Several efforts to parallelize microarchitectural simulators have been reported. All these efforts involve manual parallelization: the writer or user of the simulator has decomposed the simulator into tasks running on different processors and synchronized explicitly between the tasks. Decomposition is carried out in a top-down fashion; the most common decomposition has been to assign the simulation of each simulated processor or processor core to a host processor and either assign the simulation of the memory hierarchy to yet another host processor or simulate memory using a distributed discrete event simulator.

The Wisconsin Wind Tunnel II[67] uses a combination of binary instrumentation, direct execution, and parallel discrete event simulation on a shared-memory host to accelerate simulation of multiprocessors. The target binaries are instrumented to calculate the execution time of basic blocks and to call the memory simulator for each load or store. The memory accesses are handled by a parallel discrete event simulator. Speedups of 4.1 to 5.4 on an eight-way host are achieved; these speeds correspond to an impressive 25 host machine cycles per target machine cycle. The system is very closely tied to simulating multiprocessors and all parallelization was done explicitly. Falsafi et al.[25] analyzed the the predecessor Wisconsin Wind Tunnel System, which used a distributed memory host, and concluded that parallel simulation is more cost-effective than job-level parallelization on a network of workstations when a large model (16 or more processors) is to be simulated.

Chidester and George[17] create a parallel multiprocessor simulator for a message-passing multiprocessor by running a modified copy of SimpleScalar on each processor. Barr et al.[6] create parallel multiprocessor simulators in the Asim structural modeling system by defining new port types and changing the models to use these types. While the addition of framework support is similar in spirit to the method presented here, it still requires manual processor assignment and model changes to use the new port types. George and Cook[33] create a parallel architecture simulator hand-parallelized at the granularity of individual processor cores using the Linda shared-tuple-space programming model.

Note also that both [17] and [6] use asynchronous algorithms; processors are allowed to execute independently without agreeing on time until reaching a point where synchronization must occur; in both cases this is when a memory access reaches the first non-processor-private portion of the simulated memory hierarchy. While this speeds up simulation considerably as synchronization points are few when there are private caches, it relies upon building knowledge of when synchronization is necessary into the model. Invalidation messages can also cause problems; these messages must travel from the shared portion of the hierarchy into the private portions, which may have simulated further into the future than the message. Barr does not deal with this situation, essentially using an optimistic algorithm without any rollback. Simulators built in this fashion may not be suitable for simulating workloads where data is shared between simulated processors. Chidester uses null messages, at a cost of lower speedup due to more frequent synchronizations.

All of the aforementioned efforts have required the user or developer to manually parallelize the simulator. Not only is this time-consuming, but the difficulty of doing so limits the ability of the simulator to adapt to changes in the number of available processors. For example, a simulator of an eight-processor system parallelized for an eight processor host would be unable to take full advantage of a twelve processor host. Likewise, the simulator would not run efficiently on a system with fewer host processors. Thus automatic techniques for simulator parallelization are desirable.

4.2.3 Parallel Sampled Simulation

Another approach to parallelizing microarchitectural simulation has been to simulate different portions of the dynamic instruction stream in parallel. Nguyen et al.[69] divides an instruction trace into equal-length chunks, feeds each chunk into a detailed simulator, and combines the results for each chunk. The detailed simulation of each chunk proceeds in parallel. Because the simulation of each chunk begins with "cold" microarchitectural state, the chunks are overlapped by a fixed number of instructions; these instructions are used to warm up the later chunk.

Girbal, et al.[34] apply a similar concept in the DiST system to support execution-driven simulators; such simulators are able to simulate wrong-path execution and are thus more accurate. They observe that the warmup period required for each chunk varies and is difficult to predict a priori. They propose that chunks continue simulating instructions after nominally finishing; the statistics (e.g. cycles per instruction or cache miss rate) for these additional instructions are compared *on-line* to those obtained by simulating the first instructions of the following chunk. Once the statistics for the two simulations converge, the previous chunk may terminate simulation. Thus this technique automatically determines the warmup required by a particular chunk, improving accuracy for those requiring long warmup periods and improving simulation time for those that do not.

Others have suggested using architectural checkpoints to support simulation of samples of execution as parallel jobs; such efforts include those of Lauterbach[58] and TurboSMARTS[107]. Such techniques could also be used to provide initial state for simulating chunks in the DiST system.

All of these techniques create job-level parallelism. This parallelization is manual, though far easier to perform than the parallelizations in Section4.2.2. The automatic parallelization considered in this dissertation is complementary to parallel sampled simulation. As Section 4.1 explained, job-level parallelism might not always provide the best throughput for some systems, particularly multi-threaded or multicore processors. Future computer systems are likely to have a hierarchy of parallelism; a typical scenario might be a cluster of multicore processors. In such a situation, the ideal parallelization could be to have each separate processor handling a different chunk or sample, but the cores within each processor cooperating through an automatic parallelization of the simulator.

4.2.4 Parallel Structural Simulation Frameworks

Krishnaswamy et al.[53] have successfully parallelized VHDL simulators by using a static scheduling approach. Unlike the dynamic approach usually taken in distributed discrete-event simulation, they compile VHDL code into a statically-scheduled parallel program for a shared memory multiprocessor. Compilation is based upon the techniques of the VeriSUIF system[29]. Codeblocks extracted from the VHDL code are used as the basic tasks which are then scheduled onto individual processors. Synchronization operations are inserted as needed for notification and mutual exclusion.

Their first attempt uses list scheduling (see Section 4.4.2) with the Highest-Level-First heuristic, but they find that locking overhead causes parallel simulation with four processors to be 7.8 to 17.6 times slower than uniprocessor simulation. They add a pass after list scheduling to rearrange the schedule to avoid inserting locks and elide some notifications which are not needed due to dataflow between processors. While these improve performance for the three circuits considered, parallel simulation is still 6.9 to 17 times slower.

They then introduce a partitioning algorithm based upon fanin cones which allows work to be duplicated on different processors so that communication may be reduced. They note that if a partitioning can be found which minimizes the largest amount of work assigned to any processor, that partitioning will have maximal load balance. They then show that finding such a partition is an NP-complete problem and propose a heuristic for performing the minimization. This heuristic starts from an initial seed cone for each processor and assigns the remaining cones greedily to each processor in decreasing order of their overlap with already scheduled tasks on the processor. After forming this assignment of cones to processors they perform a phase of random attempts to swap cones to improve workload. They are able to achieve speedup for thirteen gate-level circuits, with speedups ranging from 1.09 to 2.75 for four processors.

The HASE structural simulation framework[19] is parallelized by design using a synchronous strategy. A centralized work distributor assigns the handling of events occurring at the same time to multiple processors and then waits for all of the processor to complete their assigned work before moving to the next group of events. This results in very serial execution; the only opportunities for parallel execution come when two codeblocks must run at the same delta timestep. Experiments on a highly parallel 256-processor Cray T3D yielded simulation speed not significantly better than a single processor workstation. This is because of the extreme serialization of execution, as well as the overhead of a centralized work distributor.

4.3 Providing a Parallel Microarchitectural Simulator

Parallelization of a structural microarchitectural simulator requires similar steps to those that would be taken in order to parallelize any application. This section discusses the problems that must be solved in order to parallelize structural microarchitectural simulators, as well as terminology used in this chapter.

4.3.1 Steps of Parallelization

Parallelization has four basic steps:

1. Decomposing the work of the application into tasks which will run concurrently.
2. Scheduling tasks onto processors.
3. Inserting synchronization mechanisms to provide communication and mutual exclusion as necessary.
4. Reorganizing the layout of data to match the scheduling. This may be required in systems without global address spaces or it may be desirable in order to improve the locality of references and thus cache behavior.

Achieving performance improvement and scalability as the number of processors increases depends upon maximizing the utilization of the processors. However, doing so requires an understanding of how data is accessed and organized. Thus another aspect of parallelization is understanding the dataflow internal to the application. Task decompositions or schedules which require large amounts of synchronization through communication or mutual exclusion may lead to large amounts of overhead to perform synchronization operations as well as processors idling while waiting for each other. Either outcome results in a loss of performance.

Task formation may be directed by the data structures or control flow of the program. Examples of the former approach abound in scientific code; task decomposition commonly forms tasks based upon partitioning a data structure such as a matrix. The latter approach is taken more frequently in simulation systems; each invocation of a codeblock is considered to be a separate task.

Once the task decomposition has been decided, the tasks need to be scheduled onto processors. This may be done dynamically or statically. Dynamic scheduling has the advantage of implicitly balancing load between the processors at runtime: only if there is no work available to be done do processors remain idle. A great deal of synchronization overhead is usually necessary to implement such a scheme. Static scheduling does not have this overhead, but requires careful implementation to maintain high processor utilization in the presence of varying or unknown task execution times or communication costs. Static scheduling may use either a priori knowledge of task execution times and communication costs or estimates of these factors.

Synchronization mechanisms include both communication primitives and mutual-exclusion primitives. Transfer of data between tasks executing on different processors must use communication primitives, which may simply be notifications that data is available. If exclusive access to a resource or memory location is required, mutual-exclusion primitives must be inserted. Two invocations which require mutual-exclusion primitives are termed **conflicting** invocations.

Finally, reorganizing the layout of data so that data used by tasks assigned to the same processor are kept together may be desirable. This can reduce false sharing[96] when caches are present. For parallel systems without a shared-memory programming model, this step may be required.

These parallelization steps may be performed by hand or by tools. For structural simulation frameworks, where the simulator code is generated by tools, parallelization performed by the tools is clearly desirable. However, the user could provide guidance to the tools: for example, by pre-assigning tasks to processors. As noted before, task decomposition for simulation is commonly based upon codeblock invocations.

4.3.2 Terminology and Parallel Systems

The reader may have noticed that the terms *threads* and *processes* have heretofore been avoided while describing parallelization. The reason for this was that these terms often convey certain implications about the programming model supplied by a parallel system, with threads often implying a shared address space and processes not. Some parallel systems call each task a thread. Further confusing the situation, some structural simulation frameworks call codeblocks logical processes. Others, such as VHDL, even allow programmed suspension of a logical process (e.g. waiting for a signal to become true), which actually creates multiple codeblocks for a single logical process.

This chapter continues to use the term **codeblock** as it was used in Chapters 2 and 3. The independent units of work to be scheduled will be known as **tasks**. A **thread** is defined as a portion of the parallel simulation running on a particular processor; it consists of the code to invoke a number of tasks as well as overhead for managing their scheduling and synchronization as well as the run-time state of the tasks. A processor executes only a single thread unless it is a multithreading processor.

Parallel systems are classified based upon their programming model. Programming models may be either shared-memory or message-passing. In shared-memory systems, threads may share portions (or all) of their address space and communicate implicitly through loads and stores. Synchronization mechanisms include barriers, locks to enforce mutual exclusion, and notification primitives to inform a thread that data has become available to be read in the shared memory space. In message-passing systems threads do not share an address space and communicate explicitly through send and receive primitives. Locks are not necessary as any resource requiring mutual exclusion must be in its own thread in such a model, but barriers are still provided.

This chapter also classifies parallel systems based upon the parallel hardware. Four kinds of systems are considered here: A system may be a **distributed** system, with physically independent computing nodes containing processors and memories, such as a network of workstations[4]. The system may be a **traditional multiprocessor (MP)**, with individual processors tightly coupled in a single case or rack by some sort of communication network or memory system. A parallel system may also be a **chip multiprocessor (CMP)**[70], where multiple processor cores are combined on a single chip and share some portion of the memory hierarchy. Finally, the system may be a **simultaneous multithreading processor (SMT)**[97], where instructions from multiple threads can be simultaneously executed in a processor core. Note that these systems may be hierarchically composed: the nodes of a distributed system could be traditional multiprocessors made up of chip multiprocessors with simultaneous multithreading cores.

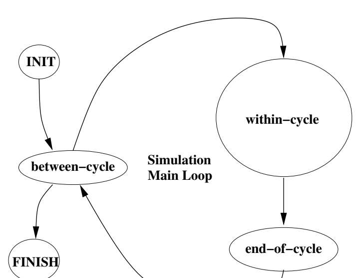
4.3.3 Parallelization in the Liberty Simulation Environment

The parallelization of simulators in the Liberty Simulation Environment was previously described in [76], and some figures and discussion in this section are taken from that work. The goals in parallelizing simulators generated through the Liberty Simulation Environment are to:

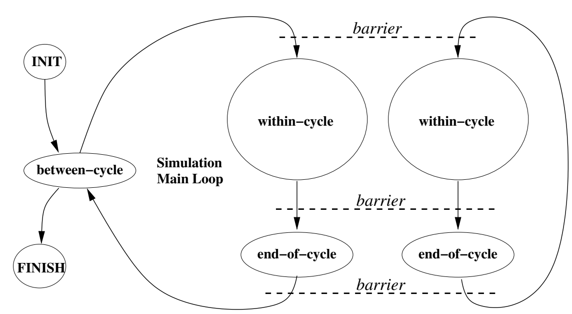
- Improve simulation speed.
- Not require changes to the model. Ideally, the user should be required to state no more than parameters of the host architecture such as the number of processors.

To meet both these goals, the framework must exploit structural information present in the model.

Parallelizing an LSE simulator is done by parallelizing the main loop of simulation. That loop was shown in Figure 2.2; it is duplicated in Figure 4.1(a). While in some systems parallelization such that different cycles are computed in parallel may be possible, in LSE, with black box codeblocks that may access state as they wish, such an approach is infeasible. Instead, the form of parallelization is that the main loop is duplicated, and individual codeblock invocations are assigned to different threads. Barriers are inserted to separate the steps of execution described in Section 2.1.3. This is shown in Figure 4.1(b).



(a) Uniprocessor



(b) Parallel

Figure 4.1: The simulation loop

This parallelization effort assumes a shared memory model. This model was chosen because LSE is not a "pure" structural system: the user is able to share state in many ways between codeblocks and instances. Removing the shared memory space inherent in the uniprocessor version would require extensive changes to many user models.

This automatic parallelization of LSE simulators is quite similar in style to that of VHDL simulators performed by [53], though the mechanisms used are quite different. Parallelization of LSE appears to be the first effort to parallelize simulators which use the Heterogeneous Synchronous Reactive model of computation.

Referring back to the steps of parallelization from Section 4.3.1, the following actions are taken as parallel simulators are generated:

Task decomposition Individual codeblock invocations within a clock cycle are considered tasks. The uniprocessor schedule of invocations is used to form a directed acyclic graph (DAG) of tasks. There is an edge between two tasks if the source invocation is intended to generate a signal which the destination invocation may read in order to generate an output and the destination invocation is after the source invocation in the uniprocessor schedule. In addition, there is an edge from any invocation which may generate a particular signal value to the last invocation which may do so; this guarantees that the final value of the signal will be known at a particular point in the schedule and allows some reduction in the locks which will be required. Separate task graphs are created and scheduled for the within-cycle and end-of-cycle steps of execution. Note that all codeblocks and their dependences are visible to the framework as part of the structure of the connection graph or dependence annotations made by the user to enhance uniprocessor scheduling.

Embedded dynamic subschedules are treated as a single task and are not parallelized. Furthermore, because they typically touch many signals and state, they are treated as serial portions of the schedule and no other tasks may run concurrently with them. Extending the parallelization to dynamic subschedules is deferred to future work.

Scheduling The distributed discrete-event simulation techniques presented in the previous section are not particularly appropriate for the parallelization of LSE simulators due to their dynamic nature. As shown in the last chapter, static partitioned scheduling is preferred to dynamic scheduling in the uniprocessor case as it generally reduces the number of codeblock invocations required. The additional overhead of selective-trace techniques was found to be detrimental to performance despite further reductions in invocations; dynamic scheduling across threads would require even greater overhead. Furthermore, the very fine task granularity would lead to even more overhead[90]. As a result, a static approach to scheduling is used.

How should codeblock invocations be divided statically between threads? There must be tradeoffs made among load balance, communication costs, the critical path of computation, and cache effects. This is essentially a problem of multiprocessor task scheduling, where the tasks are individual codeblock invocations. The scheduling problem is the core problem which needs to be solved, and is addressed in more detail in Section 4.4.

Synchronization Mechanisms The barriers which are required to separate steps of execution have already been mentioned. Binary semaphores are used in cross-thread communication to indicate that a signal value which is produced in one thread is available for use by another thread. Locks are used to ensure mutual exclusion when codeblocks may share state or access to the same location in a non-shared fashion. Note that the user is *not* responsible for insertion of any of these mechanisms; the framework does that automatically.

Potential state sharing is quite common and arises because:

- All the flow control signals for a port instance are "packed" into the same memory word.
- State may be shared by codeblocks within an instance.
- The user may define "runtime variables" and share them.
- The user may make stateful library calls such as I/O or instruction-set emulation.

The framework assumes that state sharing for the first two reasons will always occur. Sharing of runtime variables could be assumed to occur as well, but has not been in the current implementation. The final source of sharing cannot be analyzed directly by the framework because of the black-box nature of codeblocks and the library itself. As a result, the user must supply some help to the framework in the form of **conflict constraints** which say simply that two codeblocks must be mutually exclusive. These constraints are maintained in a simple text file whose name is given as a parameter to the model. This constraint file is the only way in which the goal of "no user changes" is not met, and in practice is quite simple to maintain.

Avoiding the insertion of locks where they are not required is important while inserting synchronization logic. A **lock elision** algorithm is used to remove such locks. This algorithm detects when two conflicting tasks scheduled on different threads are serialized because of communication between the threads and thus do not need locks. During scheduling the algorithm maintains a list of incoming cross-thread communication arcs for each thread sorted by their arrival time. Each arc has an associated list of times to which each thread must have advanced before the communication began. A binary search of this list allows a quick test to see whether a serializing communication has occurred. This test requires $O(\log V)$ time to perform.

Data Layout Some effort is made to place thread-private framework variables together to both increase spatial locality and reduce false sharing between threads. The main port status data structure is also organized to reduce false sharing. The impact of these minimal layout optimizations is not evaluated in this work.

4.4 The Scheduling Problem

Scheduling is a critical step; a poor schedule leads to processors remaining idle while waiting to receive data or waiting for exclusive access to a resource. Furthermore, the schedule affects locality of references and thus the cache behavior in the parallel systems considered in this dissertation, further affecting performance.

4.4.1 Static Scheduling for Structural Simulation

In this dissertation only static scheduling is considered. Dynamic scheduling of many fine-grained codeblocks seems likely to result in poor performance due to the overhead of accessing common work queues. These overheads would be at least as large as those shown for dynamic scheduling in the previous chapter and quite possibly much larger as synchronization would be required. Dynamic scheduling is also likely to provide poor cache locality.

The static scheduling problem is an instance of the multiprocessor task scheduling problem. There have been many variants of this problem studied in the literature. Table 4.1 gives the

Variant Class	Simulator Parallelization
Objective function	Makespan
Precedence constraints	Yes (computational dependence)
Resource constraints	0-1 (mutual exclusion)
Communication costs	Yes
Sequence-dependence	Yes (caching)
Cost variability	Yes
Granularity	Fine (100s of instructions)
Scale	over 10,000 tasks

Table 4.1: Variants of the multiprocessor task scheduling problem

principal dimensions used to classify variants as well as the coordinates of structural simulator parallelization within this classification scheme. The individual dimensions of the problem are:

Objective function The scheduler's goal is to minimize the time required to evaluate a single timestep. This is known in the literature as the **makespan**.

Precedence constraints Some tasks must follow other tasks because of computational dependences.

Resource constraints Tasks may require resources in addition to processor time. The sum of the resource requirements for tasks executing at any instant in time must not exceed the resources available. In simulator parallelization, these constraints arise because some tasks must run in a mutually exclusive fashion with each other. The locks which are used to ensure mutual exclusion can be modeled as resources which are either not used or completely used by a single task; this usage pattern is known as a **0-1 resource constraint**.

Communication costs The time it takes to transfer data between tasks running on different processors is non-negligible.

Sequence dependence The time it takes to execute a task depends upon the schedule. Sequence-dependence arises because of cache effects, which are dependent upon where and in what order tasks run.

Cost variability The scheduling algorithm does not know with certainty how long a task will take to run. As a result, schedules are not guaranteed to execute exactly as planned and synchronization operations must be inserted.

Granularity The granularity of tasks can be quite small; measurements taken using one model (the ***CMP02*** model from Section 4.6) suggest that most codeblock invocations execute a few hundred instructions, though a few are up to three orders of magnitude larger.

Scale There are many codeblock invocations to be scheduled; one model evaluated in this chapter has over 10,000.

The remainder of this section describes related work in task scheduling which shares some of the characteristics of structural simulator parallelization. However, no previous work has considered a problem with all of these characteristics.

4.4.2 Related Work: Precedence Constraints and List Scheduling

The simplest problem is to minimize makespan while meeting precedence constraints. The problem was originally introduced by Hu[41]: find a schedule which will minimize the time it takes to complete all tasks when the tasks have precedence constraints forming a directed acyclic graph (DAG), known as the **task graph**. An algorithm for finding the optimal solution was demonstrated by Ramamoorthy et al.[81], but has exponential complexity. The problem has been shown to be NP-complete by Lenstra[61]. As a result, the focus of most research has been on finding good heuristics.

The most well-known scheduling heuristic for minimizing makespan while meeting precedence constraints is **list scheduling**, originally introduced by Hu in 1961[41]. A general form of the list scheduling algorithm is given in Figure 4.2. A **ready list** of tasks which have not yet been scheduled and whose predecessors have already been scheduled is maintained. Tasks are selected according to some priority function from the ready list and assigned to a processor and starting time combination such that the task executes after all of its predecessors have finished and the assigned processor is not busy at that time. A particular combination of starting time and processor is called a **scheduling slot**. Note that if the priority function used at the line marked 1 is such that it produces a topological sort of the graph, the ready list can be initialized with all tasks and need not be incrementally maintained (thus removing the loop marked 3).

There are many variants of list scheduling; these variants can be classified along two dimensions corresponding to the lines marked 1 and 2 in Figure 4.2. The dimensions are:

List-Schedule(D, L, T, P)

$\triangleright$ D is the DAG of tasks to schedule
$\triangleright$ L_i is the time it takes to run (cost of) task i
$\triangleright$ T_i will be the assigned start time of task i
$\triangleright$ P_i will be the assigned processor of task i

$\triangleright$ Form predecessor counts and initial ready list
$C_i \leftarrow$ (number of predecessors of i), $\forall i \in D$
$R \leftarrow \{i | C_i = 0\}$

while $R \neq \emptyset$
  do Select a task i from R according to some priority
    $R \leftarrow R \setminus i$
    Assign task i to some processor P_i at time T_i such that
      T_i is after the finish time of each predecessor of i
      and P_i is not already busy during the interval $[T_i, T_i + L_i)$
    foreach u **in** successors of i
      do $C_u \leftarrow C_u - 1$
        if $C_u = 0$
          then $R \leftarrow R \cup u$

Figure 4.2: List scheduling

Task Priority - The original proposal calculated a metric later called **level** which was the largest sum of task costs along all paths from a task's vertex to a sink vertex in the DAG. Tasks were selected in descending order of level, therefore the strategy is known as **Highest-Level First (HLF)**. A similar strategy calculates the largest sum of task costs along a path from a source vertex to a given task vertex. When the priority is set in ascending order of this metric, the strategy is called **Smallest Co-Level First (SCF)**[2]. Many other priority functions are possible[55]. In the literature, HLF is often called HLFET where ET signifies that estimated task costs are used.

Processor/Time Assignment - Greedy strategies are very common: the scheduling slot that allows the task to run earliest is chosen. There are two methods used to implement these strategies. The first, used by Hu[41], is to maintain a global clock in a loop outside the **while** loop. A task is then not considered ready until the timestep in which all of its predecessors have finished execution. The other method is to keep track of the intervals in which each processor is used and insert the ready task into the earliest available "hole" in the schedule. The difference in methods is subtle, but does result in different schedules, as the order in which tasks are considered is different.

Adam et al.[2] evaluate several priority schemes using random task graphs of up to 200 vertices as well as task graphs derived from FORTRAN programs. They conclude that HLF significantly outperforms other priority schemes and is no more than 4.4 percent from optimal except in one case. They also look at the effects of run-time task cost variations and again find that HLF outperforms other schemes.

The execution time requirements of list scheduling depend upon the priority function and the assignment strategy. The simplest schemes such as HLF without trying to fill in holes are $\Theta(E + PV)$ where P is the number of processors and V and E are respectively the number of tasks and dependence edges between them. If attempts are made to fill holes, a straightforward implementation of HLF has execution time $O(E + PV^2)$.

An interesting characteristic of list scheduling is that a worst case bound can be given on its behavior relative to an optimal schedule. Graham[37] shows that list scheduling yields a schedule whose makespan is no more than a constant factor times the optimal makespan:

$$\frac{\omega_{list}}{\omega_{opt}} \leq 2 - \frac{1}{p}$$

where ω is the makespan and p is the number of processors.

4.4.3 Related Work: Communication Costs

Adding interprocessor communication costs to precedence constraints is a common variant of the multiprocessor scheduling problem and is considered very appropriate for distributed systems or message-passing programming models. A good survey of techniques for handling communication costs with precedence constraints is given by Kwok[55].

For shared-memory programming models and their typical implementation systems, communication costs are usually not modeled directly. This is because scheduling for these costs is typically based upon an assumption that data will be "pushed" towards consumer tasks instead of being "pulled" by them, allowing the processor where the consumer task will run to work on some other task while the data is in flight. For a shared memory system, though, delaying the consumer until the data will arrive is not possible without some sort of prefetching arrangement because the data does not start movement to the consumer's processor until the consumer is already running. However, looking at how communication costs can be handled is still instructive because, as will be shown later, sequence dependence can be modeled as a communication cost.

There are three general approaches to handling communication costs. The first approach modifies list scheduling to include these costs. The second tries to **cluster** tasks before scheduling them. The final approach duplicates tasks and assigns them to multiple threads so that communication need not occur. This approach is not considered further because duplication of tasks which may share state will lead to poor cache behavior as well as a loss of concurrency as the duplicated tasks must acquire locks.

List-based approaches

List scheduling can be modified to include communication costs. Several ways in which this has been proposed are described below.

Extended List Scheduling (ELS) Perhaps the simplest way to add communication costs is called Extended List Scheduling by Hwang et al.[42]. This heuristic is to simply use list scheduling with no communication costs and then add delays to the schedule. This is a static equivalent of simply using the schedule and allowing synchronization logic to delay tasks until inputs arise (as is needed anyway if there is variability in task length). They show that this heuristic has poor theoretical bounds on how poor the schedules can become relative to the optimal schedule with no communication and use this finding to justify development of a more complex heuristic.

Earliest Assignment Another simple way to incorporate communication costs is given by Yu[113], who calls it "Heuristic Algorithm D." This algorithm simply modifies the assignment step to choose the earliest scheduling slot such that the time is after the finish time of each predecessor of the task plus any required communication cost. The priority function can be any of those previously mentioned for list scheduling; Yu used HLF.

Earliest-Time-First (ETF) Hwang et al.[42] present the Earliest-Time-First heuristic. This heuristic modifies the priority function such that the highest priority ready task is the one which can start at the earliest time, taking into account what has already been scheduled and communication costs. Ties are broken by using the level of the tasks. The chosen task is then scheduled at the earliest time possible. The priority function need only be evaluated as tasks become ready, so the execution time is $O(PV^2)$. An interesting side effect of this prioritization scheme is that holes are never introduced in the schedule which would be filled by later assignments.

Mapping Heuristic (MH) El-Rewini and Lewis[22] describe the Mapping Heuristic. This heuristic uses a ready list which contains events. When all of the predecessors of a task are finished, a **start event** is added to the list with no communication delay. When start events are processed, the processor which minimizes the finish time of the task is chosen, based upon the communication costs which will delay its start. These costs may depend upon the current state of the schedule. A **finish event** is scheduled at that time. When finish events are processed, start events are generated for any successors which become ready. Events are always handled in increasing timestamp order, with ties broken by level with a fixed communication cost. MH provides a low-complexity way to manage a global clock with some ETF-style dynamic reprioritization. A strength of MH is its ability to incorporate network routing and contention into communication costs. The execution time is reported to be $O(P^3V^2)$ when a matrix of communication costs is maintained, though $O(E + PV \log V)$ seems possible when communication costs are fixed.

Modified Critical Path (MCP) Wu and Gajski[111] retarget the priority function to vertices which contribute to the **critical path** through the task graph with the Modified Critical Path heuristic. The critical path is the path with the highest sum of task costs and communication costs in the task graph. Vertices which are not on the critical path have some slack in when they could be scheduled if there were an unbounded supply of processors. The latest time at which a task can be scheduled without affecting the makespan is known as its **As-Late-As-Possible (ALAP)** time. The highest priority task has the smallest ALAP time, with ties resolved by the smallest

ALAP time of descendant tasks. This priority function allows the ready list to be initialized to all tasks. The execution time of MCP is $O(V^2 \log V)$.

Mobility-Directed Scheduling (MD) Wu and Gajski[111] also introduce a heuristic called Mobility-Directed Scheduling. This heuristic uses a metric called **relative mobility**. Relative mobility is the difference between the ALAP time and the **As-Soon-As-Possible (ASAP)** time – the earliest time a task can start – divided by the cost of the task. This metric is defined dynamically, changing as assignments are made and the communication costs between tasks assigned to the same processor become zero. Tasks on the critical path have a relative mobility of zero. The highest priority task has the lowest relative mobility; ties are broken by choosing a task which has no predecessors in the set of tasks with minimum relative mobility. Tasks are assigned to the lowest numbered processor which can receive them; the definition of when this can be done is complex because previously assigned tasks which have slack can be delayed to form a hole into which the new task can be placed. This is necessary because the ready list is initialized to all tasks, but the priority function does not select vertices in topological order. The execution time is $O(V^3)$. Note that this algorithm does not limit the number of processors it uses, so a post-processing step is necessary to reach a target number of processors.

Dynamic Level Scheduling (DLS) Sih and Lee[85] present Dynamic Level Scheduling. This heuristic uses a metric called the **dynamic level**. The dynamic level for each potential assignment of a task to a processor is defined as the difference between the static level (that normally used for HLF list scheduling) and the earliest time that the task could begin on that processor, taking into account what has already been scheduled and communication costs. The highest priority ready task is the one with the highest dynamic level; that task is then assigned to the appropriate processor and time. This dynamic priority function results in an execution time of $O(PV^2)$ if communication costs are considered fixed. One advantage of this technique is that it allows schedule-dependent modeling of communication costs, though this changes the runtime. When evaluated on random graphs of between 50 and 150 vertices, DLS provides up to 70% improvement in makespan over HLF. Note that if the static level is set to zero, DLS becomes ETF.

Selvakumar and Ram Murthy[83] provide a version of DLS which uses schedule holes and also schedules the communications networks.

Dynamic Critical Path (DCP) Kwok and Ahmad[54] proposed the Dynamic Critical Path heuristic. This heuristic is based upon the observation that as scheduling decisions are made, the critical path changes. DCP works similarly to MD scheduling in that the initial ready list contains all tasks and assignments may proceed out-of-order with respect to precedence. It differs from MD in the priority function and the processor selection method. The highest priority task has the lowest ALAP - ASAP, with ties broken by the lowest ASAP. These metrics are recomputed after each task assignment, with communication costs between tasks assigned to the same processor set to zero. Only processors on which predecessors of a task are scheduled plus at most one new processor are considered; the processor which minimizes the start time of the task plus the start time of the successor task with the lowest slack is chosen. Furthermore, holes are only created if there was not a large enough hole for the task already in existence. The execution time is $O(V^3)$. As with MD scheduling, the number of resulting processors is not limited and a post-processing step is required to reach a target number of processors.

Clustering approaches

Clustering is another approach to handling communication costs. An algorithm for the generic approach is given in Figure 4.3. Clustering differs from list-based scheduling in that instead of iterating over tasks attempting to schedule each one, it tries to group tasks together into clusters which will be scheduled onto the same processor, iteratively combining these clusters until some limit is reached. For some algorithms, a schedule for each cluster is incrementally maintained; for others, a list scheduling pass with processor selection constrained so that clusters remain on the same processor follows clustering. Several proposed clustering algorithms are described below.

Internalization Sarkar and Hennessy[82] present a clustering heuristic called internalization which attempts to put tasks along the critical path into the same cluster. They cluster as long as there are clusters which can be combined which would reduce the critical path length (assuming list scheduling with HLF priority) on an unbounded number of processors; the two clusters which

GENERIC-CLUSTERING(D)

$\triangleright$ D is the DAG of tasks to schedule

$\triangleright$ C is the set of clusters, each of which is a set of tasks
$C \leftarrow \{\{i\} | i \in$ vertices of $D\}$

while can continue clustering
do Choose two clusters C_j and C_k from C to merge
New cluster $C_n \leftarrow C_j \cup C_k$
$C \leftarrow (C \setminus \{C_j, C_k\}) \cup C_n$
$\triangleright$ Optionally create a schedule for tasks within C_n

return C

Figure 4.3: Clustering

maximize this reduction are chosen at each clustering step. The execution time is reported to be $O(V^2(V+E))$. A variant called EZ has also been reported[55] which simply considers edges in the task graph in descending weight order and combines clusters if the result improves the critical path length; this variant has execution time $O(E(V+E))$. To further assign clusters to processors, they consider tasks from as-yet unassigned clusters in HLF priority order and assign the corresponding cluster to the processor which results in the smallest completion time given the current assignments. This step has execution time $O(V(V+E))$.

Declustering Sih and Lee[86] introduced the declustering technique. This complex algorithm consists of five steps:

1. Form elementary clusters. Clusters are formed by cutting edges. These edges are selected by locating **branch vertices** – vertices in the task graph with multiple outedges. Branch vertices are considered in increasing static level (the level for HLF list scheduling) order. Pairs of outedges are considered in decreasing static level of their targets and zero, one, or two cuts are placed in their fanout so as to minimize the makespan. This step is reported to take $O(V^4)$ time.

2. Combine elementary clusters hierarchically. The smallest cluster in terms of total execution time is combined with the cluster with which it has the highest communication bandwidth.

Ties are resolved in favor of the cluster with the smallest total execution time. Each clustering decision is recorded in order, forming an list of clustering decisions. This is repeated until only one cluster remains. This step requires $O(V^2)$ time.

3. Decompose the cluster hierarchy. All clusters are initially assigned to one processor. The cluster decisions are then examined in reverse order. If reversing a decision to combine elementary clusters by shifting the smaller of the clusters which were combined onto another processor would reduce the schedule height obtained by list scheduling with decreasing static level used as the priority function, then shift the smaller cluster onto the processor which decreases the schedule height the most. This step requires $O(V^3P)$ time.

4. Shift some elementary clusters between processors. This step is not well-described but has two passes. The first reduces inter-processor communication along the **schedule limiting progression**, or the longest chain of dependent tasks through the schedule which does not have any slack time; the second attempts to move clusters from heavily loaded processors to lightly-loaded ones. The execution times are not given for this step.

5. Break down elementary clusters. This step attempts to break apart elementary clusters in situations where the appropriate granularity was less than that of the clusters. This is done by moving portions of the scheduling limiting progression across processors if doing so would improve the makespan. The execution time for this step is $O(N^3P)$.

Declustering explicitly schedules communication as requiring time on both the receiving and sending processors. Even though the description appears to be using a message-passing model with both send and receive operations, the authors state that declustering is designed for shared-memory systems. This discrepancy is not explained, but by modeling cache latency as time taken by the receive operation and scheduling a task to begin at the start of its receive operations, declustering could be made applicable.

Dominant Sequence Clustering (DSC) Dominant Sequence Clustering, introduced by Yang and Gerasoulis[112], places tasks on the critical path on the same processor while tracking changes to the critical path due to clustering decisions which have already been made. An explicit goal of this work was to produce a low-complexity algorithm.

DSC uses a technique very like list scheduling. There are two ready lists; the main ready list is called the **free list**. The other ready list, the **partially free list**, contains tasks for which some, but not all of its predecessors have been examined. The priority of a task in the free list is its ASAP time minus its ALAP time plus the critical path through the task graph. The priority of a task in the partially free list is its ASAP time considering only the previously examined predecessors minus its ALAP time plus the critical path. Ties are broken by choosing the task with the most outedges in the task graph.

Assignment to clusters is performed by finding a set of predecessor tasks with which the task can be merged. This is done in such a way as to ensure that the ASAP time will improve. This step may move predecessor tasks between clusters if this can be done without affecting the ASAP time of their successors. If no cluster can be found, the task begins a new cluster. The partially free list is used to deal with certain corner cases that would affect ASAP times for successors of already-examined tasks.

The restrictions that prevent the ASAP times of the successors of already-examined tasks from changing imply that ALAP need be calculated only once and ASAP can be re-calculated incrementally. This leads to a very efficient algorithm with an execution time of $O((V+E)\log V)$. In addition, DSC has been proved optimal for certain classes of task graphs. Note, however, that DSC requires a post-processing step to assign the clusters to the final number of processors.

4.4.4 Related Work: Resource Constraints

Resource constraints are a less commonly studied variant of multiprocessor scheduling. While various works, e.g. [10, 5, 43], have considered resource constraints without precedence constraints, this subsection elaborates only upon those works which have considered both kinds of constraints.

List scheduling can be used to handle resource constraints by changing the assignment step to choose the scheduling slot which yields the earliest start time while obeying both the precedence and resource constraints. Garey and Graham[32] show that the makespan will be no more than a constant factor worse than the optimal, but that constant is, unfortunately, the number of processors:

$$\frac{\omega_{list}}{\omega_{opt}} \leq p$$

where ω is the makespan and p is the number of processors.

The classic work of Fisher[28] describes microcode scheduling from the standpoint of multiprocessor scheduling with 0-1 resource constraints and advocates the use of list scheduling. He evaluates a number of list scheduling strategies, including two which form the priority function by either adding or multiplying the number of tasks which do not require use of a common resource with a task's level. These resource-aware strategies do not produce significantly better schedules for random task graphs; indeed, HLF priority produces near-optimal results. Similar approaches continue to be widely used for instruction scheduling for superscalar or VLIW processors[56].

Narasimhan and Ramanujam[68] propose a branch-and-bound technique for finding the optimal solution to the resource-constrained scheduling problem with precedence constraints. The bounds are computed using list scheduling with ALAP time as the priority function. These bounds are reported to be highly efficient, providing orders of magnitude improvement in scheduling time vs. an integer linear programming solution. The problem sizes considered are not reported, though they appear to be several hundred tasks; how efficient the technique would be on very large task graphs is unclear

4.4.5 Related Work: Sequence Dependence

Sequence dependence with precedence constraints has been considered in the context of manufacturing process scheduling, where sequence dependence arises from the need to set up new tools on multi-function machines when assigned tasks are not homogeneous. In these environments, the objective function is typically tardiness or economic value as objective functions rather than makespan. Furthermore, setup time depends only upon the last job run on a machine, not the history of all jobs as cache behavior would. Thus this work is not particularly relevant to the problem of structural simulator parallelization.

Lu[64] does provide a full integer linear programming formulation for minimization of makespan in semiconductor manufacturing scheduling. More importantly, however, Lu observes that variable communication costs can be used to model setup time, though (s)he does not elaborate more fully.

4.4.6 Related Work: Clustered Instruction Scheduling

Instruction scheduling for clustered architectures is closely related to task scheduling, although the relationship is not always acknowledged in such works. For instruction scheduling, individual instructions are tasks, the task graph is generally called the **dataflow graph**, instructions are assigned to clusters instead of processors, and individual basic blocks or traces are scheduled separately. Clustered instruction scheduling always includes precedence constraints and communication costs; task costs are fixed and usually fairly uniform. Clustered instruction scheduling does not always include resource constraints in the normal sense, though it usually supports exclusion constraints – some instruction cannot execute on a certain cluster. Sequence dependence has not been addressed directly. Several clustered instruction scheduling algorithms have been proposed.

Bottom-Up Greedy (BUG) The Bottom-Up Greedy algorithm was used in the Bulldog[23] and Multiflow[63] compilers. The idea is to start at the exit vertices of the dataflow graph and work backwards in a recursive depth-first fashion, passing back estimates of where an instruction is likely to be assigned and what effect that would have on the final schedule. These estimates are based upon the level of instructions in the graph as well as machine limitations preventing certain instructions from being on certain clusters. Predecessor vertices are considered in decreasing level order. Scheduling slot assignments are then made from the top-down as recursive invocations of the depth-first search complete; the assignment resulting in the lowest estimated schedule time given previous decisions is used. This algorithm incorporates exclusion constraints but not resource constraints.

Unified Assign and Schedule (UAS) The Unified Assign and Schedule algorithm, by Özer et al.[72], is very similar to ETF scheduling: ready tasks are scheduled in increasing order of the earliest time at which they can be begun taking into account previous scheduling decisions and data transfers. When more than one choice of cluster is possible, several different strategies are considered: a random choice, the cluster with the most predecessor instructions, and the cluster with the latest completing predecessor. A slight modification which tries to place the critical path on a single cluster is also evaluated. For several SPECint95[1] and MediaBench[59]), UAS is shown to be more effective than BUG, with choosing the cluster of the latest completing predecessor giving the best results. This algorithm does not incorporate resource constraints.

Partial Component Clustering (PCC) Faraboschi et al.[26] introduce Partial Component Clustering. Components are formed by working backwards through the dataflow graph, choosing predecessors with the highest level; a component ends when it has reached some threshold size. Smaller thresholds result in more components and potentially higher communication costs, but provide more opportunities for parallelism. Components are then assigned to clusters using an integer linear programming solver to find an assignment that minimizes a function of the load balancing and communication costs. List scheduling is then performed with constrained assignments, obeying both communication costs and resource constraints. The component assignments are then iteratively improved by either swapping components or randomly moving a component if the change improves the results of the list scheduling. They found that PCC outperforms BUG for nearly all benchmarks.

CARS The CARS algorithm by Kailas et al.[47] uses list scheduling to perform cluster assignment, register allocation, and instruction scheduling. Descending ASAP time minus ALAP time plus critical path length (i.e, the same function used in DSC) is used as the priority function; the earliest scheduling slot at which an instruction can be scheduled without violating any constraints is chosen during assignment. Resource constraints are handled explicitly, with registers being treated as resources.

Iterative Binding Lapinski et al.[57] use a modified list scheduling technique to produce cluster assignments without assigning start times. Instructions are considered in increasing ALAP order (thus providing a topological sort), with ties broken by mobility (ALAP-ASAP), then number of successors. An instruction is placed onto the cluster which minimizes the weighted sum of a penalty due to functional unit serialization (thus incorporating resource constraints), the cost of data transfer, and a penalty due to bus occupancy. Each of the penalties is computed without performing start time assignment and depends upon external parameters. Afterward, list scheduling is performed with constrained processor assignments.

Clustering is repeated with several different values of the external parameters and the clustering resulting in the best resulting schedule is used. In addition, they use an incremental improvement pass which moves single instructions between clusters if the schedule improves is used. For fairly small instruction dataflow graphs taken from scientific kernels, they show that their technique results in schedules nearly as good as those produced by PCC.

Convergent Scheduling Lee et al.[60] use a very different approach called convergent scheduling for clustered machines as well as spatial architectures. They treat the assignment of an instruction to a scheduling slot as a probabilistic process; an instruction has a certain probability of being scheduled at a particular time and location. Scheduling is organized as a series of passes which modify the distribution functions so as to obey precedence constraints (a pass much like list scheduling), obey resource constraints, or reduce communication costs. After some number of passes, each instruction is assigned to its most likely time and processor. They show improvements in makespan of up to 14% over UAS and PCC.

4.5 Solving the Multiprocessor Task Scheduling Problem

Parallelization of structural simulators requires a solution to the multiprocessor task scheduling problem with precedence constraints, resource constraints, sequence dependence, and communication costs for a large number of fine-grained tasks with variable costs. This particular combination of characteristics does not appear to have been previously studied. Therefore, new heuristics are needed to solve the problem.

Because several studies have shown that list scheduling with Highest-Level-First priority yields near-optimal results for some of the multiprocessor scheduling variants[81, 2, 28], this section first presents the results of pure HLF scheduling for the simulator. The scheduler ignores communication costs, sequence dependence, and resource constraints.

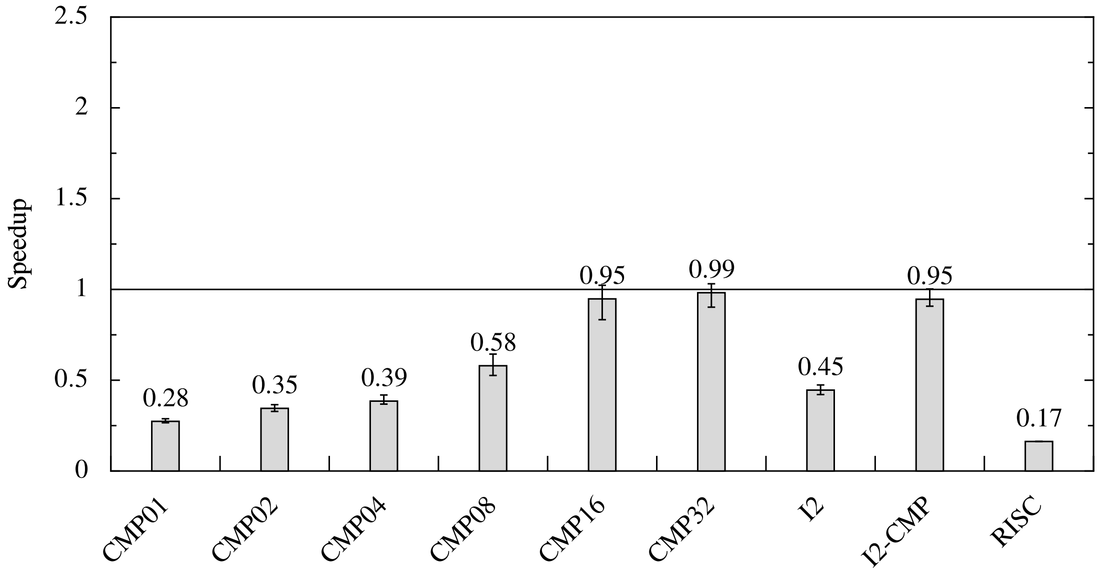


Figure 4.4: Speedup of two-threaded HLF list scheduled simulators

Figure 4.4 shows the average speedups for the simulators of a group of models when parallelized for two threads. HLF list scheduling with uniform task costs is used to generate the schedule of invocations for each thread. Speedups are measured relative to the uniprocessor simulator for each model. All simulators use non-selective-trace scheduling with full dependence information enhancement. Details of the hardware models and benchmarks will be given in Section 4.6. The error bars in all graphs in this section indicate the range of values across benchmarks.

Clearly simple HLF list scheduling does not work well; parallelization has slowed down the simulators for all models. This behavior can be examined in more detail by partitioning the speedup into two components. The first component, work concurrency or **overlap**, is the average over time of the number of processors doing work. Overlap is never less than one nor more than the number of processors. The second component, **dilation**, measures how much overhead and cache effects have changed the amount of CPU time required to perform the work of the simulator. It is calculated as the ratio of the amount of CPU time spent doing work in the parallelized simulator to the amount of CPU time spent doing work in the uniprocessor simulator. A value greater than one indicates that the time required to do work has increased upon parallelization. Time spent doing work is any time not spent waiting in synchronization operations. Details of how overlap and dilation are measured will be given in Section 4.6. Thus

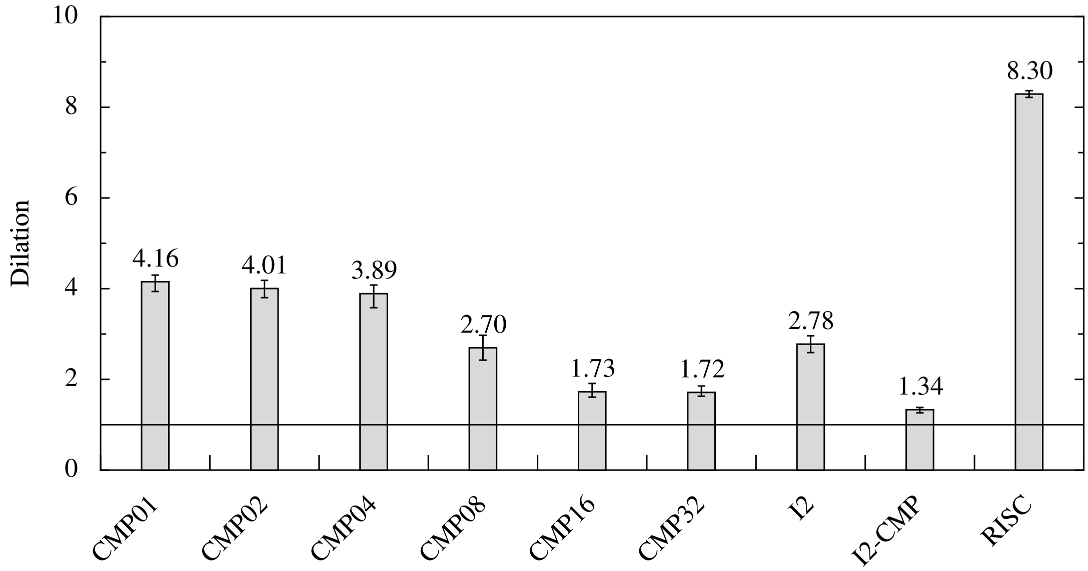
$$speedup = \frac{WallT_s}{WallT_p} = \frac{overlap}{dilation} = \left[\frac{\left(\frac{WorkT_p}{WallT_p}\right)}{\left(\frac{WorkT_p}{WorkT_s}\right)} \right] * \left[\frac{WallT_s}{WorkT_s} \right]$$

where $WallT$ indicates wall-clock time, $WorkT$ indicates aggregate CPU time spent doing work, and subscripts p and s indicate parallel and uniprocessor versions of the simulator, respectively. The final term in brackets is equal to unity because the uniprocessor simulator does not wait for other threads. Note that there is a small amount of inaccuracy because I/O time and the effects of multiprocessing on the host machine are not considered; these are verified to be very small in all experiments in this chapter, on the order of a few percent at most. If overlap is greater than dilation, then the parallel simulator is faster than the uniprocessor simulator.

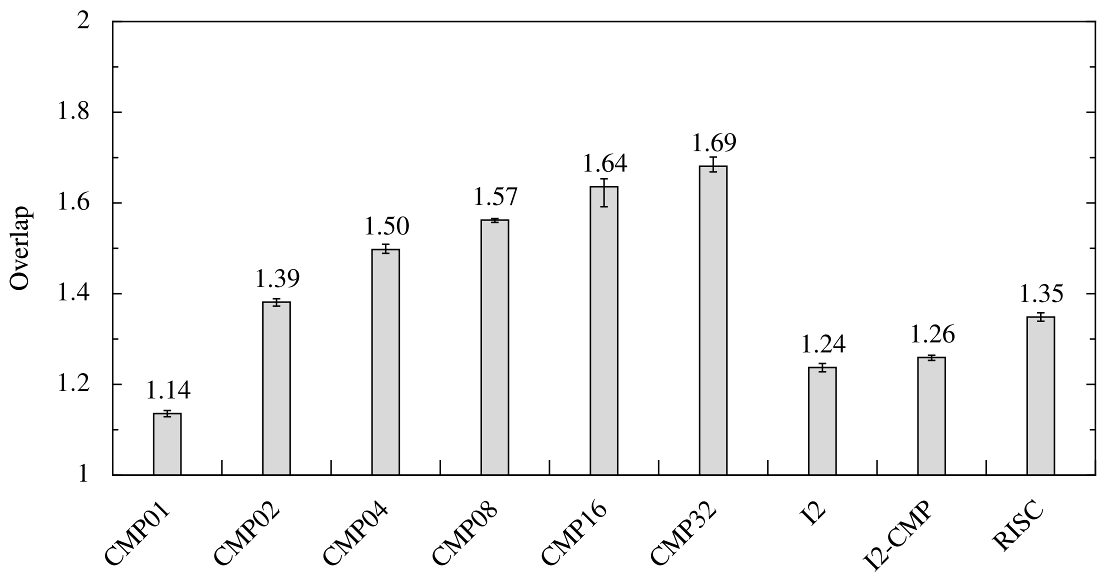
Figure 4.5(a) shows dilation for two threads using HLF list scheduling. The time spent doing work has expanded substantially. This is because of cache effects. Figure 4.6 shows the number of cache misses from the last level of the cache hierarchy per simulated cycle incurred by the parallel and the uniprocessor simulator due to data accesses. This graph shows that the number of cache misses increases upon parallelization by multiple orders of magnitude for certain models. What is happening is that the working sets of the uniprocessor simulators for the smaller models fit in the cache of a single processor.[2] When the simulator is parallelized, data which is shared between threads must now transfer between processors; this results in sharing misses. For larger models (***CMP16***, ***CMP32***, ***I2-CMP***) the original working set did not fit into a single cache and parallelization partially offsets the increase in sharing misses with a decrease in capacity misses as two caches are available. The r^2 between the logarithm of dilation and the logarithm of the relative increase in cache misses is 0.936, indicating a very strong relationship.

Figure 4.5(b) shows overlap for two threads using HLF list scheduling. The overlap also leaves something to be desired. Ideally, it would be two: at all times both threads are doing useful work. When it falls away from two, one thread is waiting. This waiting may occur at locks, for semaphores, or at barriers. Locking is the single largest component of this waiting time. Figure 4.7 shows the percentage of the overlap loss (the difference between the number of processors

[2] The ***RISC*** model is so small that the number of cache misses for the uniprocessor simulator was less than the granularity of the cache miss measurement.



(a) Dilation



(b) Overlap

Figure 4.5: Speedup components of two-threaded HLF list scheduled simulators

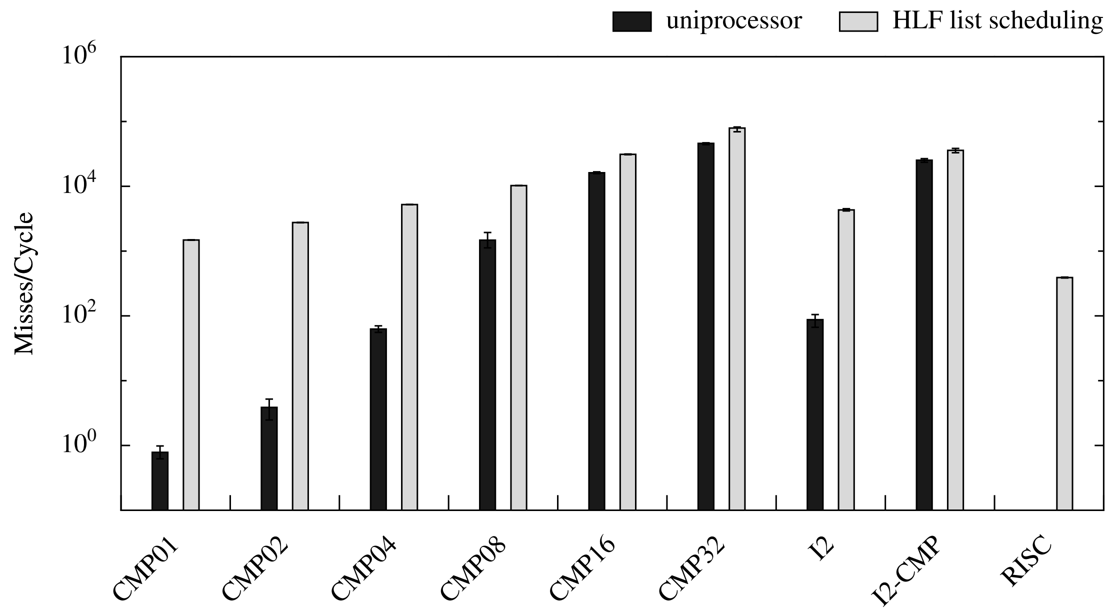


Figure 4.6: Last-level cache misses due to data accesses

and the overlap) which is due to waiting for locks. On average, about 49% of the loss is due to these waits, and the percentage of loss does not appear to depend upon model size. The remaining loss is due to communication; on average there is one communication every 5 invocations and the remaining loss correlates moderately ($r^2 = 0.51$) with the number of communication operations per invocation. The waiting for communication occurs because uniform task cost estimates describe the variable, non-uniform task costs poorly.

Attempts to predict the task costs given structural parameters such as the number of signals read and generated by an invocation have yielded very little success. This is probably due to the fact that the time a codeblock invocation takes is very dependent upon programming style independent of the interconnectivity of the codeblock graph. Codeblock invocation costs vary by several orders of magnitude with each other; furthermore, these costs are data dependent and schedule dependent. As a result, profiling of invocation times for a particular model seems likely to be the most effective prediction technique. This dissertation defers improved task cost estimation techniques to future work. Instead, it presents scheduling improvements which work when task execution time is variable or estimates are poor.

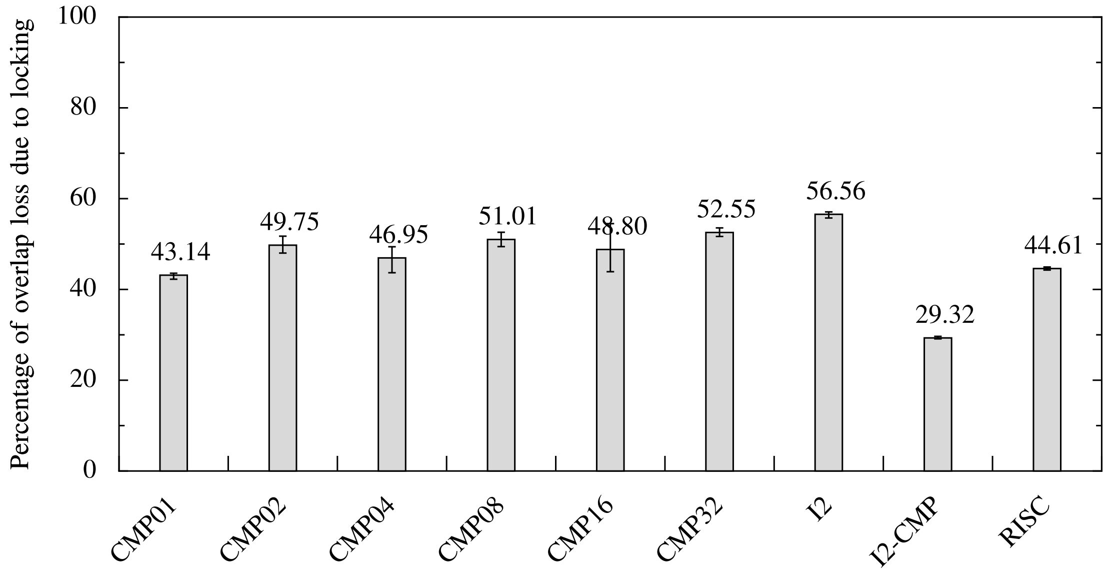


Figure 4.7: Percentage of overlap loss due to locking

There are therefore two areas upon which this dissertation focuses to improve the scheduling: reducing lock wait time and improving cache locality. The remainder of this section introduces two heuristics for reducing lock wait time and two heuristics for improving cache locality.

4.5.1 Reducing Lock Wait Time via Lock Mitigation

Time spent waiting for locks accounts for a significant fraction of the loss of overlap and thus loss in performance. Locks introduce resource constraints into the scheduling problem. Their impact on performance is greatly increased by task cost variability; variability causing an unplanned concurrent execution of conflicting tasks results in unexpected serialization of the threads.

Fisher's techniques[28] and others from instruction scheduling might be applicable here because they simply do not assign tasks to scheduling slots which will violate a resource constraint. However they are based upon having and knowing fixed task costs and thus do not take into account the variability of task costs and the likelihood that the estimates are poor. Narasimhan and Ramanujam's branch-and-bound algorithm[68] has a similar problem and potentially suffers from large runtimes on the large task graphs stemming from structural simulators. Krishnaswamy's method[53] of removing locks by rearranging the schedule after completing list scheduling does not take into account load balancing or the critical path.

In this dissertation I propose two new lock mitigation heuristics derived from list scheduling called **lock avoidance** and **lock reprioritization** which are able to cope with task cost variability and large task graphs while still taking into account the critical path and load balance.

Lock avoidance

The premise of lock avoidance is very simple: try to choose scheduling slots which do not require locks. Recall that locks are required whenever conflicting tasks might run concurrently. This analysis for lock insertion must be done without assumptions as to the speed of the tasks. Thus avoiding assignments which require locks is naturally resilient to variability or error in task cost estimation.

The lock avoidance heuristic is a variant of list scheduling. The HLF priority function is used, but the assignment step is different. A task is assigned to the earliest scheduling slot such that no lock is required. If no such assignment can be made, the assignment is made without considering locking.

Lock avoidance might seem to tend to assign every invocation in a group of conflicting invocations to the same thread, thus harming load balance. However, the majority of conflicts do not require locks because the lock elision algorithm is able to detect and remove these locks. This allows many conflicting invocations to be distributed among threads.

The execution time of this heuristic is $O(PV^3 \log V)$. The additional factor of $V \log V$ over normal list scheduling is due to the check to see whether locks are required for each potential conflict.

Lock rescheduling

When the lock avoidance heuristic cannot find a scheduling slot which does not require a lock it chooses the slot allowing the earliest start time. This can result in situations where tasks requiring the same lock are scheduled close together in time and the threads serialize on that lock at run time. The lock rescheduling heuristic tries to avoid run-time serialization by selecting start times which separate or skew in time tasks which require the same lock. To ensure that these tasks are actually separated at run time, other ready tasks are scheduled between them.

The lock rescheduling heuristic uses list scheduling with dynamic priorities, much like the ETF algorithm. The priority function groups the potential scheduling slots for each task into three groups: those requiring no locks, those requiring a lock which is not “close”, and those which require a close lock. The groups are checked in this order and the slot with the earliest start time in the first non-empty group is chosen. “Close” is defined as having a conflicting invocation overlapping in scheduled time. To accommodate task cost variability and errors in estimation, the estimated time is multiplied by some factor when making this check. For the experiments in this work, this factor was set to 10.

The execution time of the lock rescheduling heuristic is $O(PV^4 \log V)$. The reason for the additional complexity compared to lock avoidance is that the priority function needs to be evaluated for each ready task at each step. A dynamic programming technique is used to reduce these evaluations in the average case, but the asymptotic worst-case complexity is unaffected.

4.5.2 Improving Cache Locality via Clustering

Improvements in cache locality promise to improve the performance of parallelization. Cache behavior changes the task costs in a complex fashion dependent upon the schedule and the patterns of access to shared memory implied by the schedule. This is a form of sequence dependence which is far more complicated than that which is normally considered and there is little to be gleaned from prior work on sequence dependence.

Cache behavior can in part be modeled as a communication cost; any time data must be transferred between threads, there is some delay. Some prior work allows varying communication costs. However this delay is unusual in that it cannot be overlapped with computation in the receiving thread, as the communication is initiated by the receiving thread itself. Moreover, communication cost only models cache effects for signal data; it does not model cache effects due to internal data shared between codeblocks from the same module instance. For this reason, the standard list scheduling techniques for dealing with communication cost such as ELS, ETF, MH, MCP, DLS, MD, or DCP are not appropriate.

List scheduling could be modified to understand that the choice of processor affects not just communication costs but also task costs by using either priority functions or assignment rules which consider the finish time of the task being assigned. MH would be the easiest to adapt in this fashion as it already creates events for finish time. However, lock mitigation heuristics must be also be combined with these modifications, making for a very complicated overall heuristic.

Task duplication techniques such as those used in [53] are undesirable because duplicated tasks may share state, making cache behavior worse, and because duplicated tasks would need to acquire locks. These locks would probably not be elided; the point of task duplication is to create independent cones of logic. The additional locks would reduce the benefit of duplication. In addition, as task cost estimation is currently poor, the likelihood of duplicating a very expensive task seems inordinately high.

Partitioning techniques proposed for distributed logic simulation have limited applicability. Several attempt to improve metrics only of interest for optimistic distributed algorithms. Others are concerned only with load balance. Those that attempt to minimize communication in compiled logic simulation, such as depth-first, greedy levelized, and annealing, are sufficiently similar to list scheduling techniques that they suffer from the same issues.

A more straightforward approach is to use clustering techniques. A clustering is performed and then list scheduling is used afterward with the choice of threads constrained. The lock reduction heuristics can be used directly during list scheduling. Note that even though the choice of thread will be constrained, the choice of holes is not, so lock reduction techniques still have some space in which to operate.

Previous clustering heuristics such as internalization, declustering, and DSC have considered only communication costs. Because not all cache behavior can be modeled well as communication costs, using these heuristics directly is not likely to be effective. I therefore propose two new heuristics based upon clustering which address the cache behavior inside instances.

Instance-Based Clustering (IBC) The first new heuristic, Instance-Based clustering (IBC) is based upon the assumption that shared data within module instances is large. It therefore ensures that all invocations of codeblocks from a particular module instance are always assigned to the same thread. This assignment may also improve instruction cache behavior if codeblocks share functions as well as data.

INSTANCE-BASED-CLUSTERING(D)
    $\triangleright$ D is the instance graph with edges weighted by bandwidth

  Assign each instance in D to a unique cluster; call the set of clusters C

  Calculate list of inter-cluster connections L sorted by
      descending bandwidth between the associated instances

  while $L \neq$ EMPTY **and** $|C| >$ NUMTHREADS
    do $(C_j, C_k) \leftarrow$ first element of L
      Delete first element of L
      if (bandwidth between C_j and C_k) $\geq$ LIMIT
        then Combine C_j and C_k to form new cluster C_n
          $C \leftarrow (C \setminus \{C_j, C_k\}) \cup C_n$
          Calculate bandwidths from C_n to other clusters and update L,
            maintaining the sort order

  $\triangleright$ Perform mapping
  Initialize thread loads to 0
  foreach C_i in S, ordered by descending size
    do Assign C_i to thread t with minimum load
      Add size of C_i to load of t

  return cluster assignments

Figure 4.8: Instance-Based clustering

Figure 4.8 gives the algorithm for Instance-Based clustering. Note first that the instance graph is used instead of the task graph; instances are the basic granularity of clustering. Edges are weighted with the communication bandwidth in either direction between the instances. In the present implementation, the bandwidth is the number of signals connected between the instances. Initially each instance is in its own cluster.

Clusters are merged in descending order of bandwidth. Merging is stopped when there are no merges left to consider or the number of clusters is the same as the number of threads. A merge is not carried out if the bandwidth is less than some limit; 10 has been used in for the experiments in this dissertation. This limit is intended to prevent load balancing failures in the final steps of the merging as the inter-cluster bandwidth becomes small. The bandwidth between clusters is updated at each merge.

Clusters are then mapped to threads in descending order of size in a load-balanced fashion. The size is measured as the number of codeblocks in the cluster. Each cluster is assigned to the thread with the minimum load so far. This technique is similar to that of [94].

This heuristic ignores completely the precedence constraints and critical path of the task graph. Therefore, it may suffer from poor overlap even as it improves dilation. However, the heuristic is quite simple to implement. The execution time is $O(E^2 logE + PV)$ where E is measured on the instance graph. Note also that while E is $O(V^2)$ in general, instance graphs are usually quite sparse. Thus the execution time is in practice closer to $O(V^2 logV + PV)$.

Instance-Aware Dominant Sequence Clustering (iDSC) The second new heuristic, Instance-Aware Dominant Sequence Clustering (iDSC), uses the DSC technique[112] to produce a list of clusters. Predicted cache misses due to inter-task communication are modeled as communication cost. DSC is chosen because it tries to take into account both load balancing and the critical path while still maintaining a very low computational complexity. Instance awareness is added in the post-processing step which maps clusters to threads.

iDSC-MAPPING(C)
  ▷ C is the set of clusters, each of which is a list of tasks

  For each task,
  Initialize thread loads to 0
  foreach C_i in S, ordered by descending size
      do Assign C_i to thread t with minimum load which already has
              a codeblock from an instance which has a codeblock in C_i.
              If no such thread, choose the thread with minimum load.
          Add size of C_i to load of t

  return cluster assignments

Figure 4.9: iDSC mapping

The mapping algorithm is shown in Figure 4.9. It is very like the mapping performed for Instance-Based Clustering, except that it attempts to map the cluster to a thread where instances which have codeblocks in the cluster have already appeared. As there are usually many more clusters than threads, this results in clusters containing the same instances being pulled into one thread. Load balance may be compromised by doing so. This mapping also ignores the precedence constraints and critical path, as did the mapping of Instance-Based Clustering.

The DSC step has execution time of $O((V+E)\log V)$ while the mapping step has execution time of $O(PV)$.

4.6 Traditional Multiprocessor Evaluation

This section presents an evaluation of the effectiveness of the lock mitigation and clustering heuristics on a traditional multiprocessor. The heuristics are evaluated separately and then in combination, with a detailed look at their interactions.

4.6.1 Methodology

All experiments are carried out using the Liberty Simulation Environment. Partitioned scheduling with non-selective-trace scheduling is used to generate a uniprocessor schedule, which is then used as the basis for simulator parallelization. All dependence information enhancement mechanisms are used. Each parallelization is evaluated against a baseline of a uniprocessor simulator for the same hardware model.

Models

The evaluation uses nine different processor models. Four of these models were used in Chapter 3 to evaluate uniprocessor scheduling techniques. Six of the models are from a family of chip multiprocessor models of different sizes and are particularly interesting in that they show the effects of model size upon parallelization. The framework is used to generate a simulator for each scheduling technique for each model. The performance of these simulators running various benchmarks and input sets is measured. Performance measurements are taken from a single simulation run; simulation lengths were chosen to achieve wall-clock time of at least 5 minutes (though 20 data points are between 4 and 5 minutes of wall-clock time).

Model	Instances	Signals	Within-cycle tasks	Benchmarks
CMP01	81	1043	445	3 Splash2 kernels * 2 inputs
CMP02	158	2088	891	3 Splash2 kernels * 2 inputs
CMP04	312	4182	1787	3 Splash2 kernels * 2 inputs
CMP08	620	8370	3579	3 Splash2 kernels * 2 inputs
CMP16	1236	16754	7171	3 Splash2 kernels * 2 inputs
CMP32	2468	33522	14355	3 Splash2 kernels * 2 inputs
I2	209	3869	1812	3 SPEC INT 2000
I2-CMP	530	13411	6830	3 Splash2 kernels
RISC	49	407	89	2 kernels

Table 4.2: Models and input sets

The models are summarized in Table 4.2. The table indicates the number of module instances, signals, and codeblocks for each model. Each of the models and their input sets is described in more detail below:

CMP01 - CMP32 This is a family of six models of chip multiprocessors having from 1 to 32 processors. The authors were David A. Penry and Julia S. Chen. The microarchitecture is tiled, i.e. there are a varying number of tiles, each containing a processor with first level instruction and data caches, a portion of a distributed second-level cache, and connections to an on-chip routing network. Each tile has an independent channel to memory. The processors are configured to be scalar and in-order and to use the PowerPC instruction set. This model is very similar to the models used in [16]. The four-processor instance was used in Chapter 3.

For each model, three benchmarks from the SPLASH-2 benchmark suite[110] are used. For each benchmark two inputs are used: a small problem instance and a large problem instance. The binaries were compiled using gcc 3.4.1 with flags `-g -O2 -lm -static`. A sampling technique is used: one slice of execution is run. This slice begins after some number of instructions have completed on the first core. This number is chosen so that all threads have begun execution by that time. The slice ends when the first core has completed a further fixed number of instructions. The input and sampling parameters are given in Table 4.3.

The six selected benchmark/input combinations from SPLASH-2 elicit varying architectural statistics from the models, indicating that they provide some variety of internal model

Benchmark	Input parameters	Sample Start	Sample Length
CMP01 - 1 tile			
cholesky.small	-p1 -C32768 lshp.O	25,000,000	80,000,000
fft.small	-p1 -m16 -n1024 -l5	30,000,000	80,000,000
radix.small	-p1 -n262144	20,000,000	80,000,000
cholesky.large	-p1 -C32768 tk29.O	500,000,000	80,000,000
fft.large	-p1 -m24 -n1024 -l5	8,000,000,000	80,000,000
radix.large	-p1 -n33554432	2,000,000,000	80,000,000
CMP02 - 2 tiles			
cholesky.small	-p2 -C32768 lshp.O	25,000,000	40,000,000
fft.small	-p2 -m16 -n1024 -l5	30,000,000	40,000,000
radix.small	-p2 -n262144	20,000,000	40,000,000
cholesky.large	-p2 -C32768 tk29.O	500,000,000	40,000,000
fft.large	-p2 -m24 -n1024 -l5	8,000,000,000	40,000,000
radix.large	-p2 -n33554432	2,000,000,000	40,000,000
CMP04 - 4 tiles			
cholesky.small	-p4 -C32768 lshp.O	30,000,000	10,000,000
fft.small	-p4 -m16 -n1024 -l5	30,000,000	10,000,000
radix.small	-p4 -n262144	20,000,000	10,000,000
cholesky.large	-p4 -C32768 tk29.O	500,000,000	10,000,000
fft.large	-p4 -m24 -n1024 -l5	8,000,000,000	10,000,000
radix.large	-p4 -n33554432	2,000,000,000	10,000,000
CMP08 - 8 tiles			
cholesky.small	-p8 -C32768 lshp.O	40,000,000	5,000,000
fft.small	-p8 -m16 -n1024 -l5	30,000,000	5,000,000
radix.small	-p8 -n262144	20,000,000	5,000,000
cholesky.large	-p8 -C32768 tk29.O	500,000,000	5,000,000
fft.large	-p8 -m24 -n1024 -l5	8,000,000,000	5,000,000
radix.large	-p8 -n33554432	2,000,000,000	5,000,000
CMP16 - 16 tiles			
cholesky.small	-p16 -C32768 lshp.O	40,000,000	1,500,000
fft.small	-p16 -m16 -n1024 -l5	30,000,000	1,500,000
radix.small	-p16 -n262144	20,000,000	1,500,000
cholesky.large	-p16 -C32768 tk29.O	500,000,000	1,500,000
fft.large	-p16 -m24 -n1024 -l5	8,000,000,000	1,500,000
radix.large	-p16 -n33554432	2,000,000,000	1,500,000
CMP32 - 32 tiles			
cholesky.small	-p32 -C32768 lshp.O	60,000,000	250,000
fft.small	-p32 -m16 -n1024 -l5	30,000,000	250,000
radix.small	-p32 -n262144	20,000,000	250,000
cholesky.large	-p32 -C32768 tk29.O	1,000,000,000	250,000
fft.large	-p32 -m24 -n1024 -l5	8,000,000,000	250,000
radix.large	-p32 -n33554432	2,000,000,000	250,000

Table 4.3: Input and sampling parameters for the ***CMP*** family of models

behavior. For example, for ***CMP-04*** the number of instructions completed per cycle ranges from 2.20 to 2.47 across the six benchmark/input combinations. Branch prediction rates range from 87% to 100%, while the first-level cache read miss rate varies from 0% to 44%.

I2 This is the validated Itanium® 2 model used in Chapter 3. The benchmarks and input sets from that chapter are used.

I2-CMP This is the two-way chip multiprocessor model based upon the Itanium® 2 model which was used in Chapter 3. The same benchmarks and input sets are used that were used there.

RISC This is the simple single-issue in-order processor from [36] which was used in Chapter 3. The benchmarks and input sets used are from that chapter.

Because the CMP models each have six benchmark and input set combinations, all result graphs in this chapter report average results for each model rather than individual combinations of benchmarks and input sets. Error bars indicate the minimum and maximum values of the metric in question across the benchmarks and input sets.

Compilation and Evaluation Systems

All simulators are compiled using gcc 3.4.4 with the default compilation flags provided by LSE's `ls-build` script. All simulations are run on a four processor system; each processor is an AMD Opteron™ Processor 846 running at 2.0 GHz. This system has 1 megabyte of on-chip L2 cache per processor and 6 gigabytes of memory. The system runs Fedora Core release 3, with kernel version 2.6.12-1.1376_FC3smp. The system architecture is actually a Non-Uniform Memory Architecture (NUMA); processors have individual memory controllers and the cost of a memory access depends upon the location of the data. The OS, however, is not configured to perform any page placement to take advantage of the latency differences, so a reasonable assumption is that access time is evenly distributed across physical memory used by a program. Indeed, for this particular system the difference in access time between a local access and a remote access is only around 10%.

Wall-clock time is measured using `/usr/bin/time`. The wall-clock time includes time to start the simulation binary, initialize, finalize, and perform all I/O including reading of simulation state checkpoints. In addition, `oprofile`, a statistical profiling tool, is used to sample the program counter every 100,000 clock cycles and every 100,000 cache misses going to memory. This sampling is done independently for each processor. The time spent doing useful work is computed as the percentage of total clock samples across all processors for which the counter is not in one of the synchronization wait functions times the wall clock time times the number of processors. The approximate number of cache misses is computed using the total number of cache samples.

Parameters for heuristics

Each task is assumed to take 146 cycles; this is the average cost per invocation across all tasks of the uniprocessor simulator for the ***CMP02*** model. For lock rescheduling, the margin used for calculating whether conflicting tasks are close in time is 10 times the task cost. For IBC, inter-cluster bandwidth is the number of signals connected between module instances in the clusters. For iDSC, the cost in cycles of inter-task communication from task i to j running in different threads is taken to be: $112 * (1 + NS_{ij})$ where NS_{ij} is the number of signals generated by i and used by j. 112 cycles was chosen to be roughly the cost of a remote memory access.

4.6.2 Lock Mitigation

This subsection evaluates the effectiveness of the lock mitigation heuristics at reducing waiting for locks. Figure 4.10 shows the average speedup for each model's simulator when parallelized onto two threads using no lock mitigation, lock avoidance, or lock rescheduling. Speedups are measured relative to the uniprocessor simulator for each model. The error bars show the highest and lowest speedups for each benchmark.

Both lock avoidance and lock rescheduling increase simulator performance, allowing parallelization to achieve speedup for the larger models. This improvement takes place for two reasons: the heuristics have achieved their objective of reduced lock wait time and they have had a serendipitous side effect of improved cache locality.

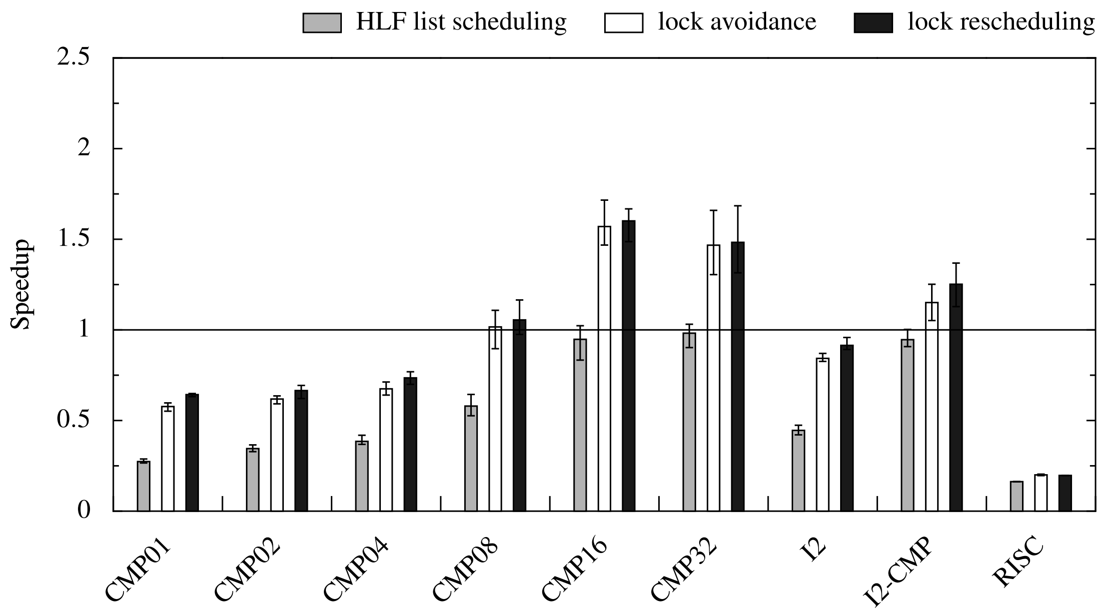


Figure 4.10: Speedup for two-threaded parallelization with lock mitigation

	none	avoid	resched.
CMP01	0.73	0.04	0.05
CMP02	0.73	0.06	0.00
CMP04	0.74	0.07	0.03
CMP08	0.74	0.10	0.04
CMP16	0.74	0.11	0.05
CMP32	0.74	0.11	0.04
I2	0.82	0.14	0.06
I2-CMP	0.86	0.14	0.08
RISC	0.58	0.21	0.08

Table 4.4: Lock acquisitions per task with lock reduction heuristics for two threads

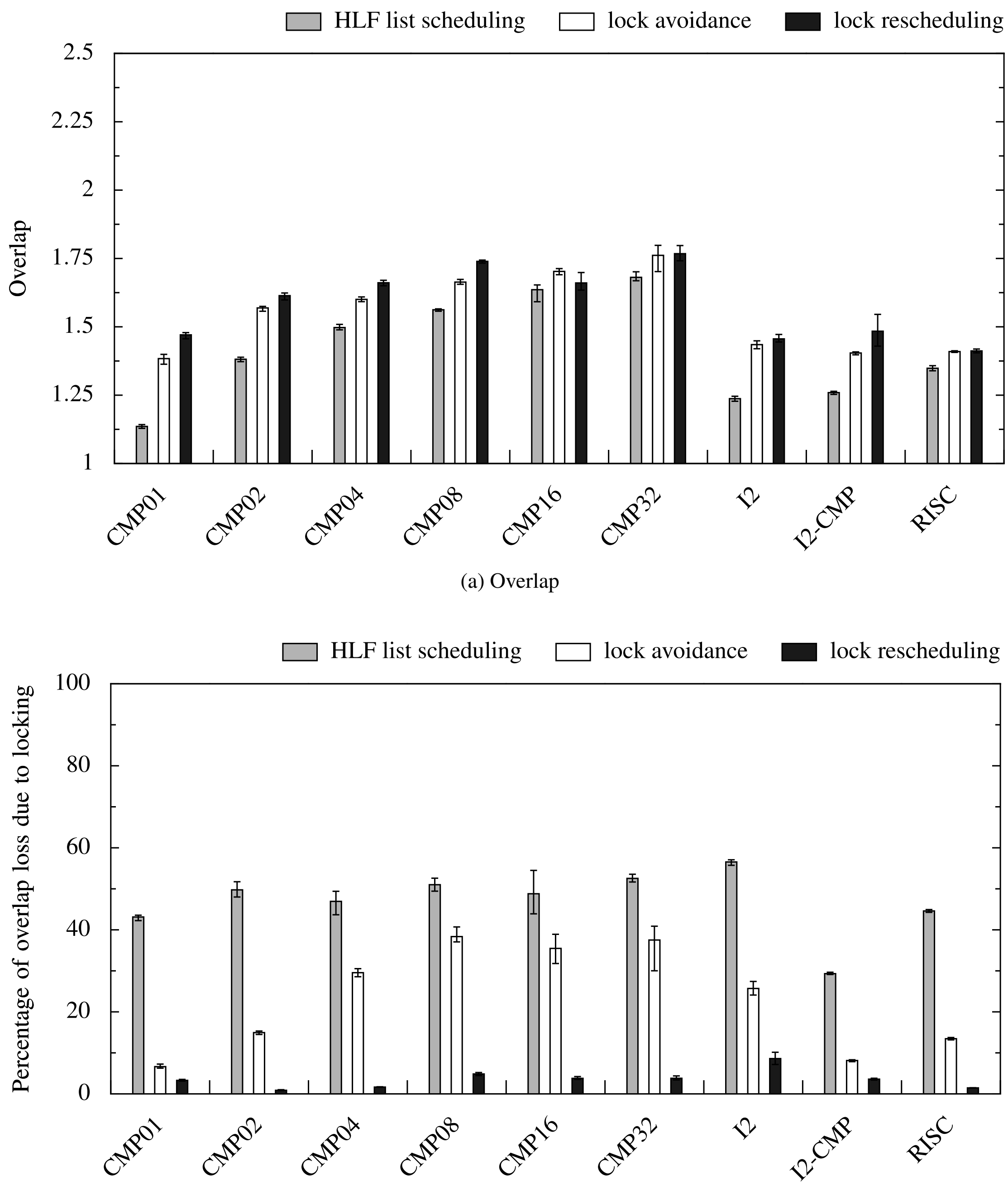
Table 4.4 presents the number of lock acquisitions per task. Locking is reduced considerably by either heuristic. Lock rescheduling usually inserts fewer locks than lock avoidance because the scheduling of a task requiring the lock is delayed in many case until the lock can be elided. Note that not using some sort of lock wait time reduction technique results in very frequent lock acquisitions: from 58 to 86 lock acquisitions per 100 tasks.

This reduction in lock acquisitions results in a decrease in waiting for the locks. Figure 4.11(a) shows the average overlap achieved by each heuristic while Figure 4.11(b) shows the percentage of lost overlap attributed to locks. The percentage of time spent waiting that is attributed to locks decreases dramatically; with lock rescheduling very little such waiting occurs. Overall waiting time as measured by overlap does not improve quite so much, indicating that some wait time at locks is being replaced with wait time at semaphores or barriers. Nevertheless, there is always improvement, and lock rescheduling is nearly always more effective at increasing overlap than lock avoidance. In the two cases where it is not (***CMP16*** and ***CMP32***), time saved spent waiting at locks is being transmuted into even more time spent waiting for semaphores.

The lock mitigation heuristics also have the side effect of improving cache locality, leading to less dilation of the simulator's work time. This is shown in Figure 4.12(a), which shows the dilation for each heuristic and Figure 4.12(b) which shows the number of last-level cache misses caused by data accesses. Lock mitigation has reduced the number of cache misses, resulting in lower dilation. This is because choosing threads which do not require locks tends to cause tasks from the same instance to execute in the same thread. For the ***CMP16*** model, the number of misses is reduced to nearly the level of the uniprocessor simulator and dilation approaches one.

The combination of the improvements in overlap and dilation leads to meaningful speedup for the larger models. For these models, the addition of lock reduction heuristics has made parallelization effective, yielding up to 80% parallel efficiency. Furthermore, lock rescheduling is nearly always more effective overall, though the difference is small enough for larger models that using the less complex and faster-running lock avoidance heuristic for these models would be acceptable.

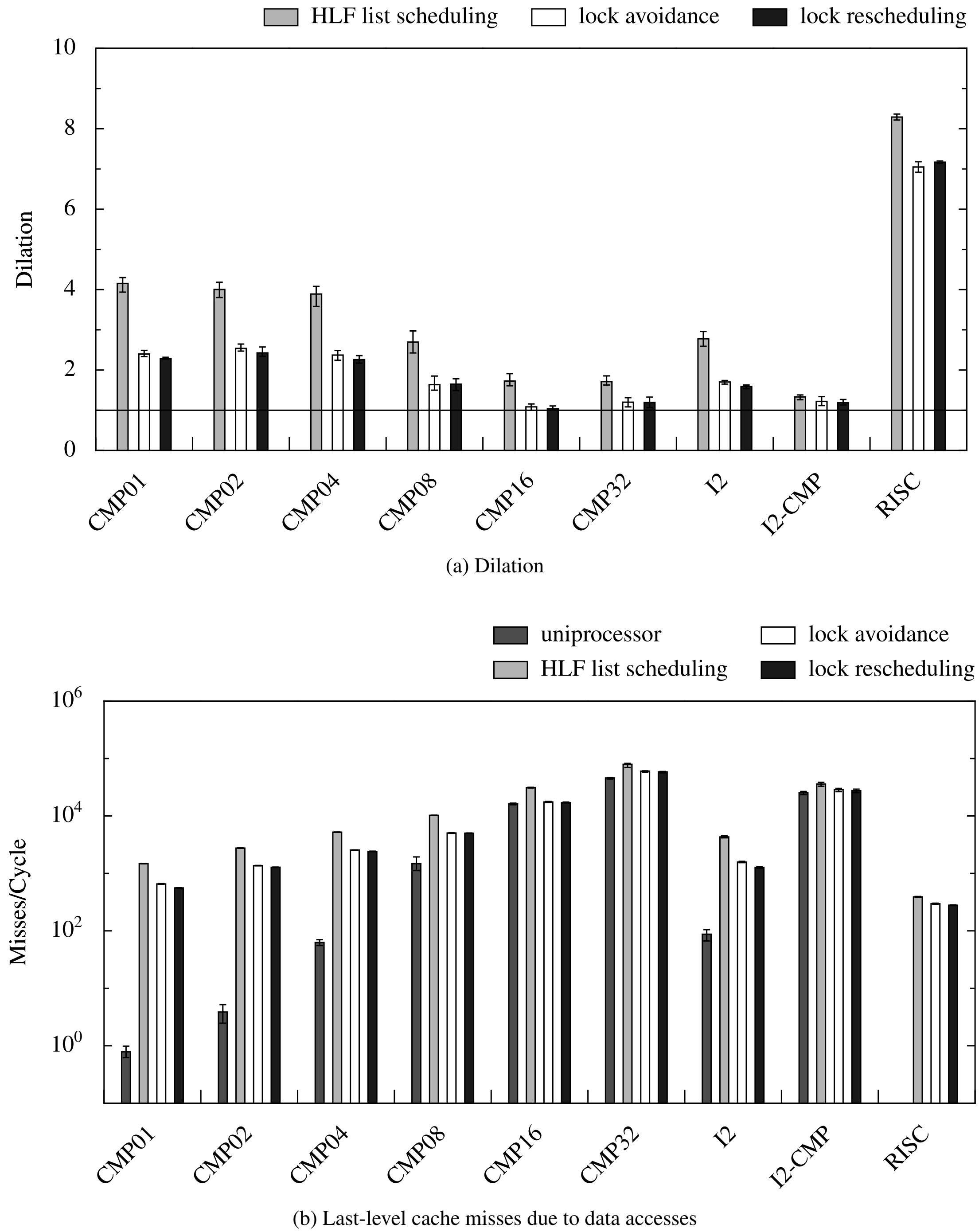
When parallelization is carried out onto four threads, reducing lock wait time should be more important as there are more opportunities for conflicts between tasks. Figure 4.13 shows the



(a) Overlap

(b) Percent of lost overlap due to locks

Figure 4.11: Effects of lock mitigation on overlap for two threads



(a) Dilation

(b) Last-level cache misses due to data accesses

Figure 4.12: Effects of lock mitigation on dilation for two threads

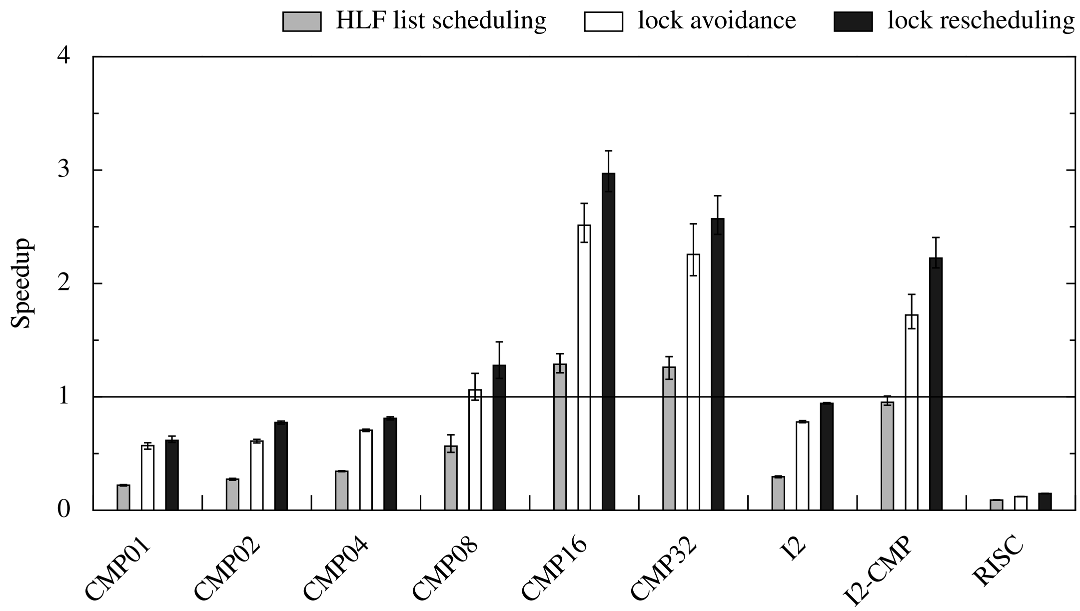


Figure 4.13: Speedup for four-threaded parallelization with lock mitigation

average speedup for each model's simulator when parallelized onto four threads using no lock mitigation, lock avoidance, and lock rescheduling. Speedups are again measured relative to the uniprocessor simulator for each model, with error bars showing the highest and lowest speedups for each model.

	none	avoid	resched.
CMP01	0.81	0.13	0.08
CMP02	0.78	0.13	0.03
CMP04	0.77	0.15	0.06
CMP08	0.77	0.14	0.08
CMP16	0.77	0.12	0.08
CMP32	0.77	0.13	0.07
I2	0.86	0.17	0.10
I2-CMP	0.90	0.30	0.14
RISC	0.75	0.58	0.37

Table 4.5: Lock acquisitions per task with lock reduction heuristics for four threads

As in the two-threaded case, lock avoidance and lock rescheduling both improve simulation speed considerably. However the difference between lock rescheduling and lock avoidance is far more visible. Table 4.5 indicates the number of lock acquisitions per task. As before, locking is reduced considerably by either heuristic and without the heuristic there is very frequent locking.

Figures 4.14 and Figure 4.15 show the effects of lock mitigation on overlap and dilation respectively. Lock mitigation has increased overlap as it did in the two-threaded case, but lock rescheduling is now always clearly superior to lock avoidance. Dilation has also decreased due to improvements in cache locality, leading the ***CMP16*** and ***I2-CMP*** models to experience true reductions in the work time. This occurs because the increase in sharing misses due to parallelization is being balanced by a decrease in capacity misses for these large models.

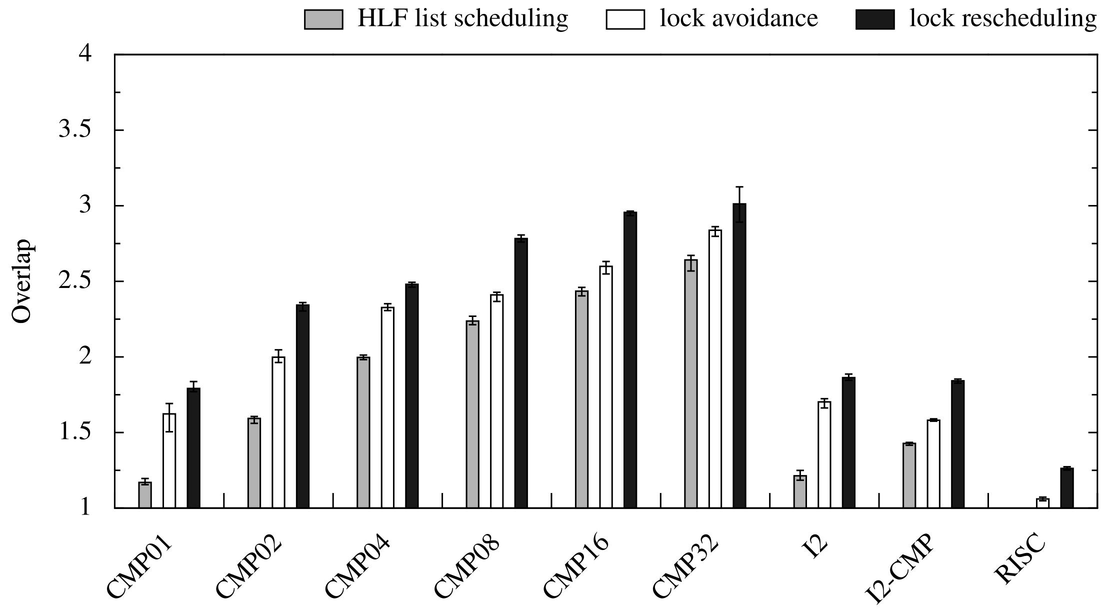
With four threads, parallelization has improved performance for the four largest models. Three of them – ***CMP16***, ***CMP32***, and ***I2-CMP*** – have enough performance to recommend four-threaded parallelization, achieving up to 74% parallel efficiency.

The CMP family of models show an interesting behavior: the speedup increases as the model size gets larger until speedup peaks at 16 tiles. The reason for this is related to the dilation behavior of the model, which also peaks at 16 tiles. This phenomenon will be elaborated on more fully in the next subsection.

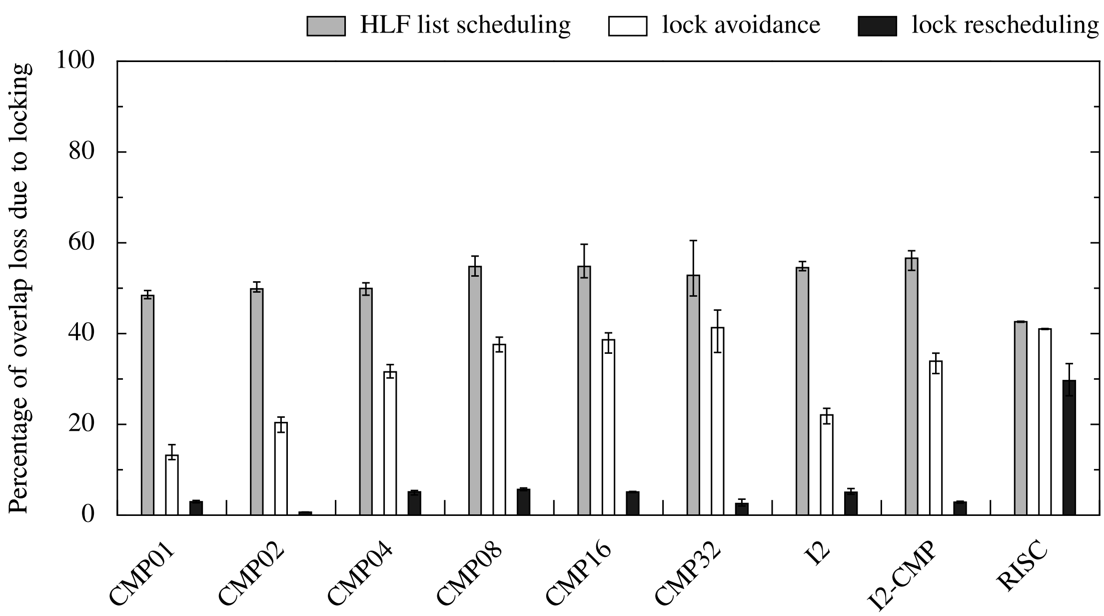
To summarize, either lock avoidance or lock rescheduling are very effective at reducing the number of locking operations and the amount of time spent waiting at locks. This leads directly to improvements in overlap and indirectly to reductions in dilation. The overall result is that either heuristic is effective at improving parallel simulation performance. Lock rescheduling is more effective than lock avoidance when there are four threads.

4.6.3 Clustering

This subsection evaluates how well the Instance-Based clustering (IBC) and Instance-Aware Dominant Sequence Clustering (iDSC) heuristics improve cache locality. Figure 4.16 shows the speedup for each model's simulator when parallelized onto two threads using no improvements, IBC, and iDSC. Speedups are measured relative to the uniprocessor simulator for each model. The error bars show the highest and lowest for speedups for each benchmark. After clustering, lock rescheduling is used in each case with processor assignments constrained by the clustering. Thus the "no clustering" case is the same as the "lock rescheduling" case from the previous subsection.

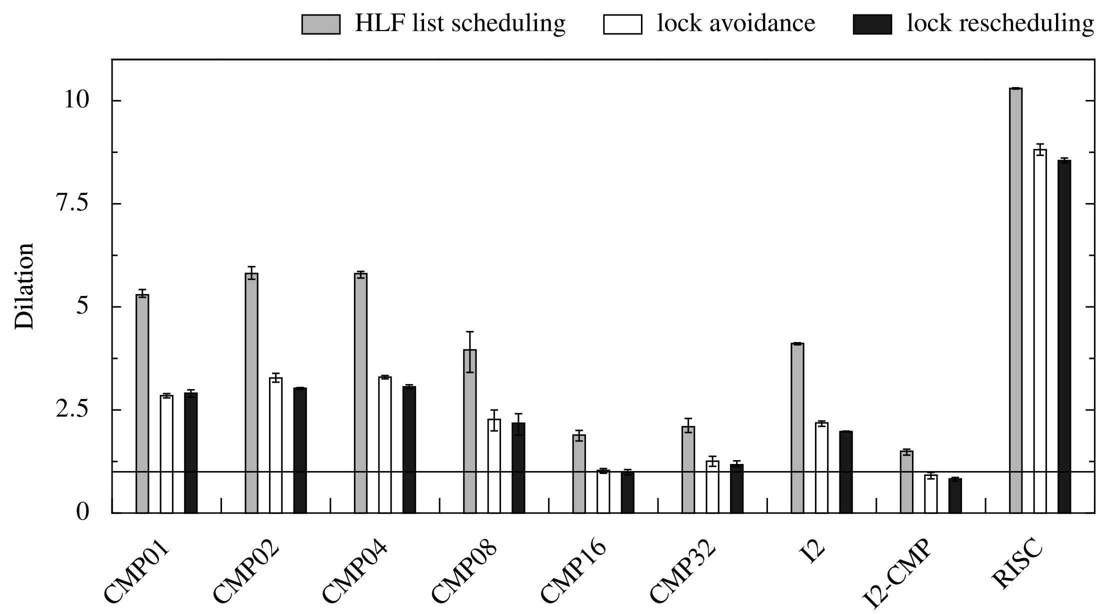


(a) Overlap

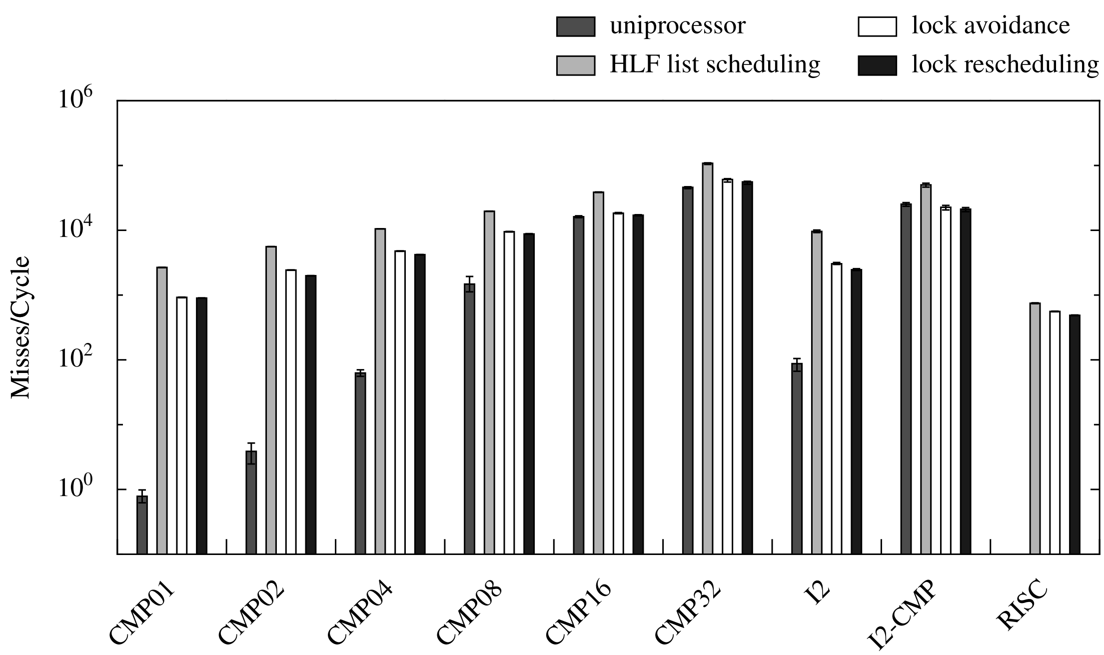


(b) Percent of lost overlap due to locks

Figure 4.14: Effects of lock mitigation on overlap for four threads



(a) Dilation



(b) Last-level cache misses due to data accesses

Figure 4.15: Effects of lock mitigation on dilation for four threads

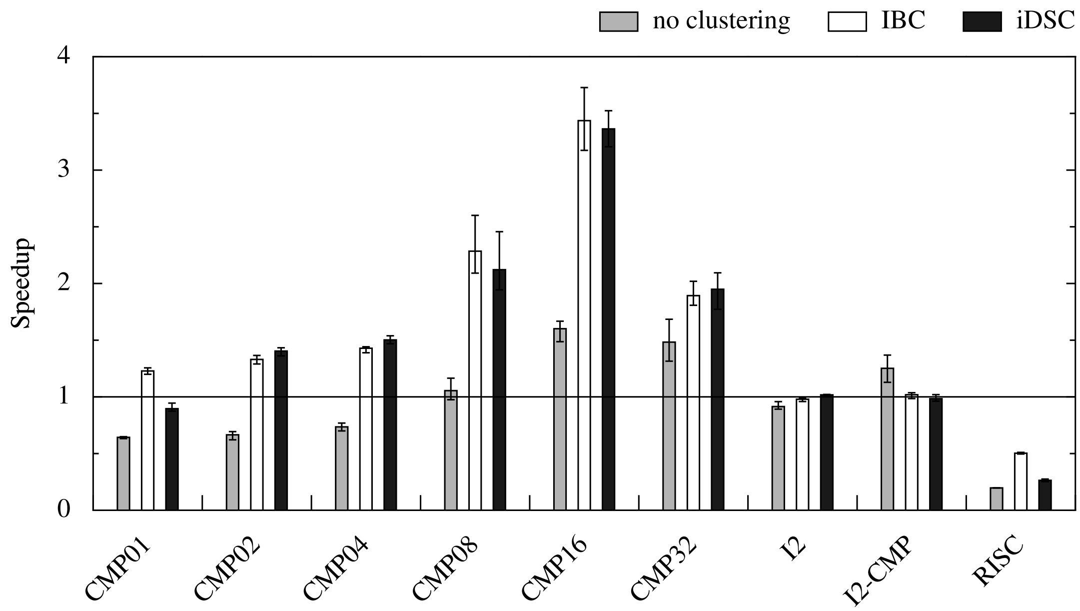
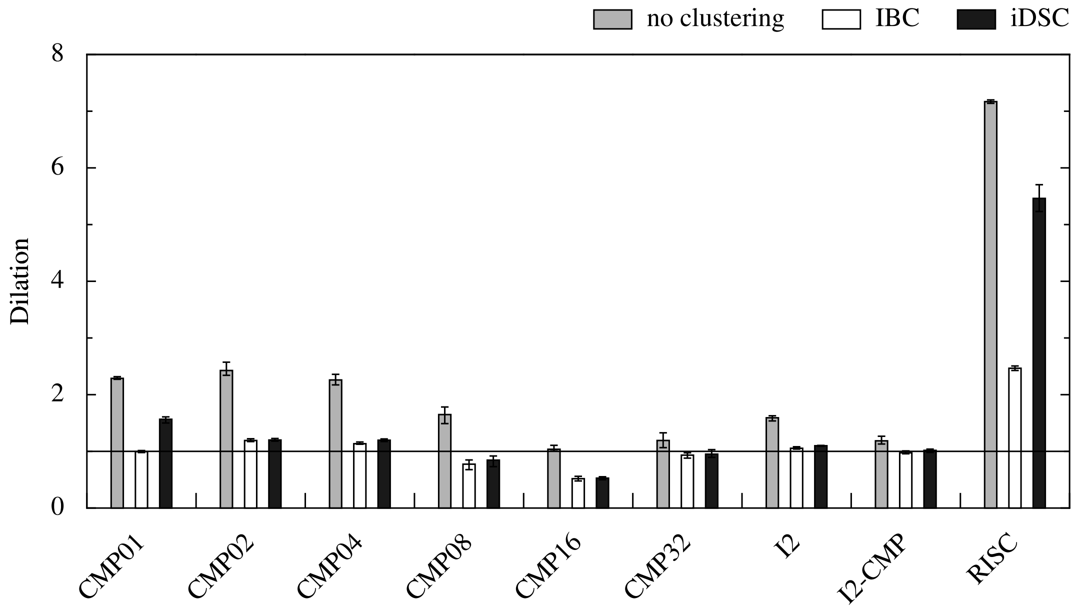


Figure 4.16: Speedup for two-threaded parallelization with clustering

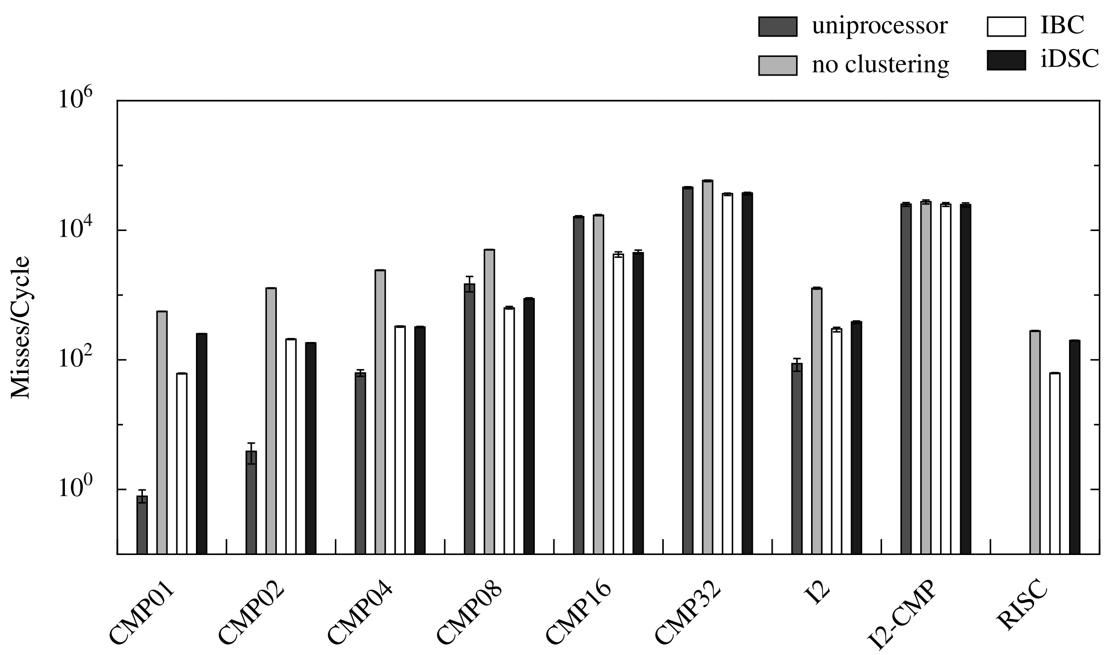
Both IBC and iDSC improve simulator performance except for one model, the ***I2-CMP*** model, which will be discussed shortly. Parallelization is now effective for all of the CMP models. Indeed, superlinear speedup is obtained for the ***CMP08*** and ***CMP16*** models. Nearly all this improvement takes place because cache locality has been improved.

Figure 4.17(a) shows the dilation for each clustering heuristic while Figure 4.17(b) shows the number of last-level cache misses caused by data accesses. Clustering has reduced dilation considerably by decreasing the number of cache misses. For the ***CMP08***, ***CMP16***, ***CMP32***, and ***I2-CMP*** models, clustering has reduced the number of cache misses to less than the number of misses in the uniprocessor simulator and dilation is less than one. The decrease in capacity misses is larger than the increase in sharing misses, resulting in better overall cache behavior and achieving a reduction in the work time required.

The ***CMP32*** model is an interesting case. The dilation increases from the ***CMP16*** model; cache locality does not improve as substantially for the ***CMP32*** model as it did for the ***CMP16*** model. This occurs because of the relationship between working set size and cache size. For the uniprocessor simulators, the number of misses increases by a factor of approximately 12 for every doubling in size of the model until the step from 16 to 32 tiles. At the final doubling step



(a) Dilation



(b) Cache misses

Figure 4.17: Effects of clustering on dilation for two threads

between 16 and 32 tiles, the misses increase by only a factor of 3. This is reasonable behavior; as the working set size increases for a fixed size cache, misses increase until the cache is eventually overwhelmed and captures little locality. Further increases in the working set result in decreasingly smaller increases in cache misses. In the case of the simulator what happens is that as the model grows, less and less locality extends across time steps; eventually the first access to every signal or data value in each cycle becomes a miss and then repeated accesses at widely separated times in the schedule become misses.[3] If the working set can be decreased for the same amount of total data, cache misses decrease and the curve of misses versus total data size shifts to the right. When simulators are parallelized with clustering, the working set of each thread is decreased, causing the growth in misses to be shifted to the right. Eventually, the new working sets saturate their individual caches. This behavior is evident in the data; parallelization shifts the curve to the right: for the first few doublings in size the cache misses grow at a rate of only 1.3 for each doubling (the rate is lower because each model starts with sharing misses), but between 8 and 16 tiles they grow by 6.7 times and between 16 and 32 tiles they grow by 8.6 times. When the size grows to 32 tiles, the total misses of the parallel simulator nearly "catch up" to the uniprocessor simulator as the working sets saturate the individual caches. This explanation is further supported by the miss data shown in Figure 4.20(b) for four-threaded parallelization, which shows yet a further shift to the right in the curve; misses are just beginning to grow rapidly in the ***CMP16*** to ***CMP32*** step.

What this all implies is that there is a "sweet spot" in the relationship between model size and number of processors. When too few processors are used, not all the potential for reduction in capacity misses is achieved. When too many are used, the increase in sharing misses outweighs the additional cache capacity. But when the right number of processors is chosen, cache misses are minimized and the work time contracts significantly. This in turn can contribute enormously to simulation speed, providing superlinear speedup. With superlinear speedup, *both* throughput and latency are improved by parallelization.

IBC usually reduces cache misses slightly more than iDSC, leading to lower dilation in nearly all cases. Even when the two techniques lead to approximately the same number of L2 cache

[3]For the ***CMP32*** model, just the data structures for holding signal values require at least 218KB, or nearly one fourth of the L2 cache.

misses, IBC still has a small dilation advantage, indicating that IBC may have advantages for L1 cache behavior, or possibly instruction cache behavior.

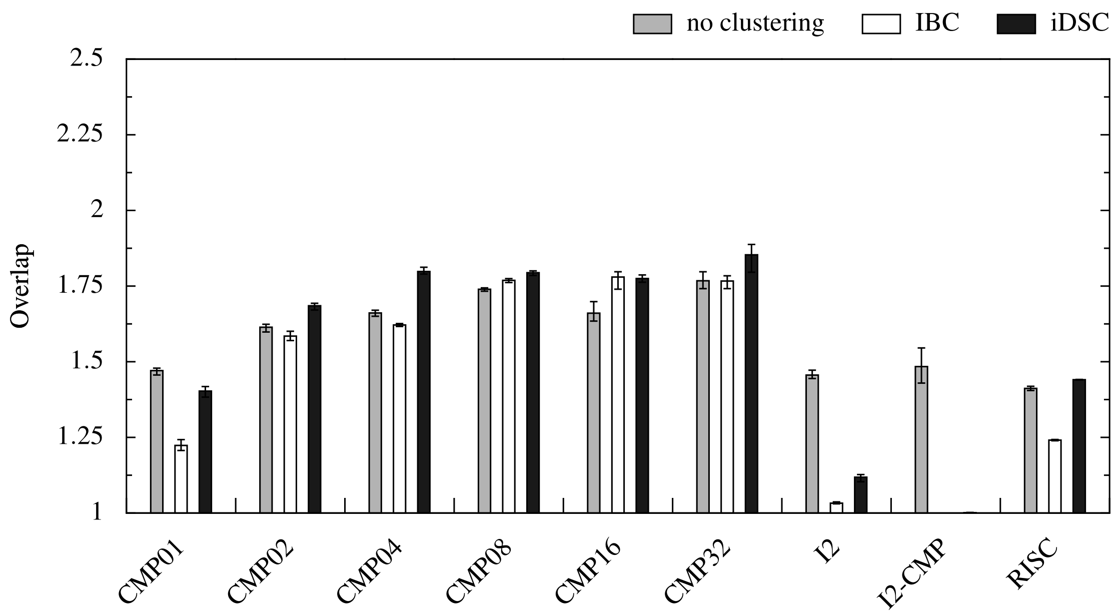
Of course, dilation is only one component of performance. Figure 4.18 shows the effects of clustering techniques upon overlap. These results are ambiguous; clustering usually disimproves overlap, but at other times it improves overlap slightly, particularly for iDSC. Such a result is reasonable; the iDSC might have taken into account critical path better than normal lock rescheduling would have. This improvement occurs in spite of an increase in the percentage of lost overlap spent in locking. However, the total number of locks introduced has barely changed.

The ***I2*** and ***I2-CMP*** models are very interesting cases. As mentioned before, clustering reduces the performance of the ***I2-CMP*** simulator. The ***I2*** model does not see performance improvement from parallelization, despite being about the size of the ***CMP04*** model. In both of these cases the clustering algorithms have "broken down," creating significant load imbalance. The load imbalance for each model, defined as the maximum number of tasks per thread minus the minimum number of tasks per thread divided by the total number of tasks, is given in Table 4.6. Both the ***I2*** and the ***I2-CMP*** models experience high load imbalance; ***CMP01*** has non-trivial load imbalance as well. The IBC heuristic causes more load imbalance than iDSC.

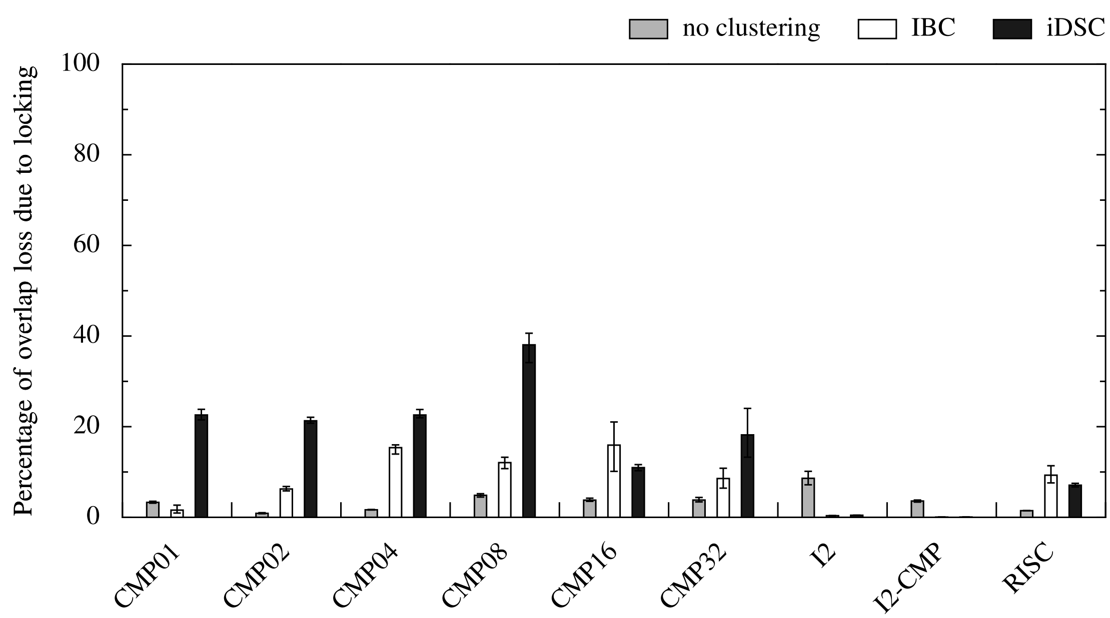
Is load imbalance really a problem? Apparently for the ***CMP01*** model some amount of imbalance was a good decision; performance is nearly twice what it would be without clustering. Performance improved for the ***I2*** model and disimproved for the ***I2-CMP*** model when clustering was used. That there could be models for which parallelization is just an entirely poor decision seems reasonable. This suggests that the algorithms need further tuning and/or development and provides an intriguing problem for future work.

	none	IBC	iDSC
CMP01	0.22	60.90	17.30
CMP02	0.11	1.01	0.79
CMP04	0.06	0.73	0.06
CMP08	0.03	8.69	0.03
CMP16	0.01	6.74	0.01
CMP32	0.01	7.52	0.01
I2	0.11	94.37	78.37
I2-CMP	0.70	95.75	92.09
RISC	3.37	55.06	3.37

Table 4.6: Percent load imbalance for two-threaded parallelization with clustering



(a) Overlap



(b) Percent of lost overlap due to locks

Figure 4.18: Effects of clustering on overlap for two threads

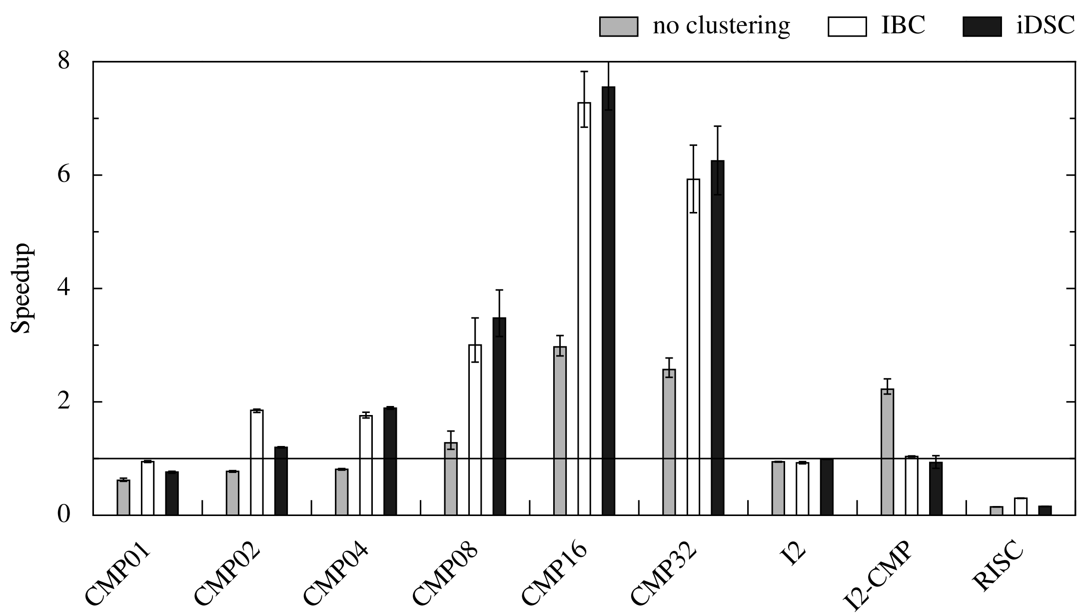


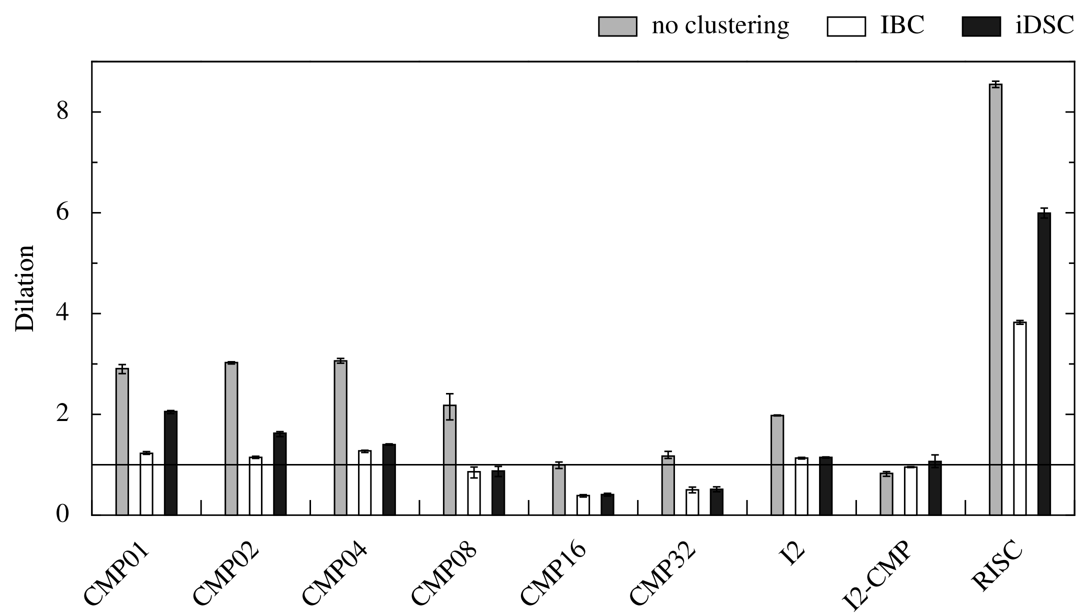
Figure 4.19: Speedup for four-threaded parallelization with clustering

Overall, clustering leads to performance improvement for all models but the ***I2-CMP*** model. The combination of lock rescheduling with clustering has made parallelization of all of the CMP family of models beneficial; for 8 and 16 tiles superlinear speedup is achieved. The iDSC heuristic leads to slightly better results unless superlinear speedup is obtained, in which case IBC performs better. Given the lower scheduling-time complexity of iDSC, neither technique is clearly more desirable than the other.

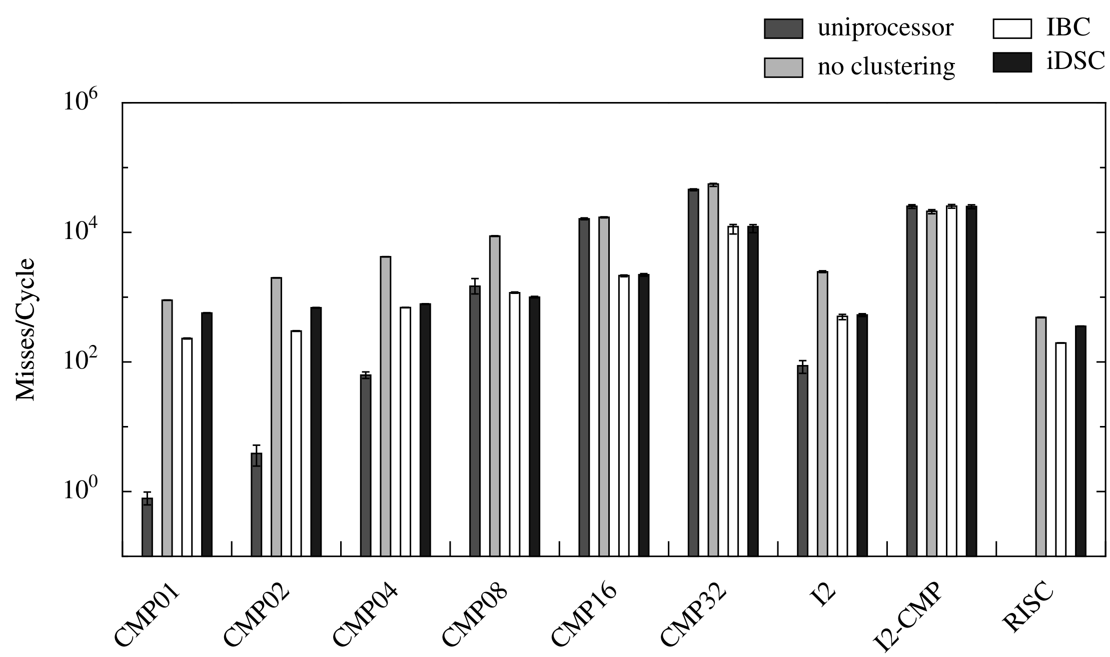
When parallelization is carried out onto more threads, clustering should be more important because the opportunity for sharing misses increases and also give more opportunity for improvement because the working set can be further divided among threads. Figure 4.19 shows the average speedup for each model's simulator when parallelized onto four threads using no lock mitigation, lock avoidance, and lock rescheduling. As always, speedups are measured relative to the uniprocessor simulator for each model, with error bars showing the highest and lowest speedups for each model. Lock rescheduling is used in all cases.

The results are quite similar to those when two threads were used. Clustering improves performance except for the ***I2*** and ***I2-CMP*** models. Most of the CMP family of models benefit from parallelization. A difference is that the ***CMP01*** model now does not benefit from parallelization;

four threads appears to be too many for this small model. In general, the smallest models (***CMP01*** and ***RISC***) have performed worse with four threads. The largest models (***CMP08***, ***CMP16***, and ***CMP32***) perform much better and continue to achieve superlinear speedup.



(a) Dilation



(b) Cache misses

Figure 4.20: Effects of clustering on dilation for four threads

Figure 4.20 presents the effects which clustering have upon dilation and cache misses when there are four threads. The same trends as were present with two threads are evident, with two exceptions.

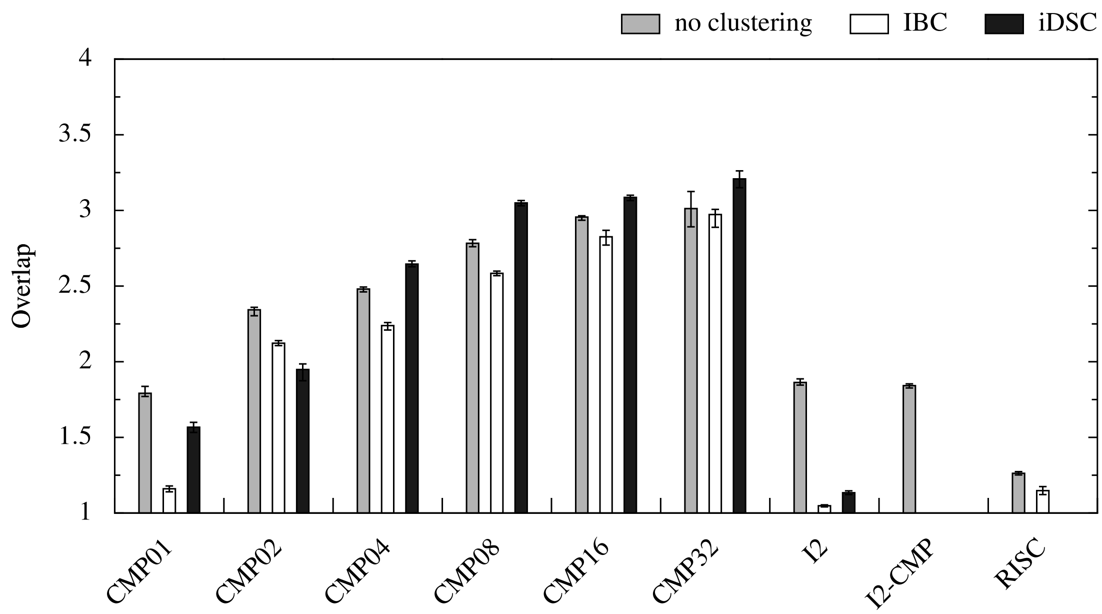
First, the ***I2-CMP*** model now does not experience reduction in dilation due to clustering. This is because the number of cache misses is increasing slightly with clustering. This in turn is happening because, just as in the two-threaded case, load balancing is poorer with clustering. As a result, the caches for three of the threads are underutilized.

Second, the ***CMP16*** and ***CMP32*** model both have lower dilation with clustering than was present with two threads. This is due to the greater reduction in cache misses which having four caches among which to partition the working set has allowed. The working sets of the threads are now further from saturating the caches for the ***CMP32*** model.

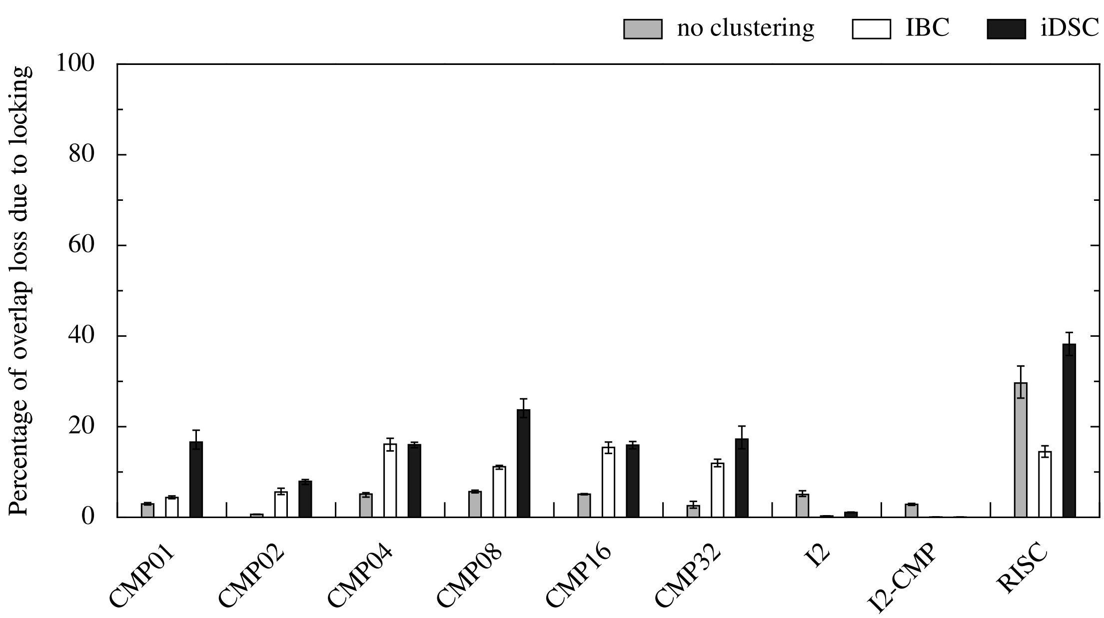
Figure 4.21 presents the effect of clustering upon overlap when there are four threads. As with two threads, the results are difficult to interpret. However iDSC does improve overlap and IBC disimproves it; the additional understanding of the dynamic critical paths through the task graph which iDSC has and IBC does not have probably explains this. The ***RISC*** model behaves unusually in that the fraction of overlap attributable to locking and the number of tasks which require locks decreases considerably with IBC. This is because the load balance is very poor and fewer locks are required with so much imbalance. Load balance for all the models is shown in Table 4.7; it has generally worsened as the number of threads has increased.

	none	IBC	iDSC
CMP01	11.24	77.30	5.39
CMP02	0.34	30.86	34.01
CMP04	0.11	4.25	1.01
CMP08	0.06	11.09	0.03
CMP16	0.03	6.53	0.01
CMP32	0.01	6.42	0.01
I2	21.69	96.63	86.59
I2-CMP	4.93	97.23	94.76
RISC	8.99	70.79	4.49

Table 4.7: Percent load imbalance for four-threaded parallelization with clustering



(a) Overlap



(b) Percent of lost overlap due to locks

Figure 4.21: Effects of lock mitigation on overlap for four threads

Overall, with four threads clustering leads to improvement for most of the models. Superlinear speedup is obtained for the largest models. The iDSC heuristic is now clearly favored for large models, other than ***I2-CMP***.

To summarize, clustering techniques in combination with lock rescheduling are effective at improving cache locality and reducing dilation. These techniques can provide significant speedups for most models. The speedup increases as the number of threads increases as long as the model is large. Instance-Aware Dominant Sequence Clustering is better for smaller models or for more threads. Instance-Based Clustering is better for large models if few threads are used. Further research is warranted into determining a priori how many threads should be used, tuning of heuristics, and whether parallelization should be attempted at all for some models.

4.6.4 Interactions

The previous two sections have shown that lock mitigation techniques are effective at reducing the lock wait time, thus increasing overlap, and that clustering techniques are effective at improving cache locality, thus reducing dilation. The previous sections have also shown that the effectiveness of these techniques is dependent upon the size of the model and the number of threads. However, they have not shown whether there are interactions between the techniques.

The interactions can be determined by looking at the combinations of techniques together. Further interactions can be seen by examining these combinations for two and four threads and for different sizes of models (or even each model) separately. Table 4.8 shows the average speedup of each combination of techniques for each combination of model size (small, medium, or large) and number of threads (two or four). The small models are ***CMP01***, ***CMP02***, and ***RISC***; the medium models are ***CMP04***, ***CMP08***, and ***I2***; and the large models are ***CMP16***, ***CMP32***, and ***I2-CMP***.

The presentation of these interactions does not follow the format of traditional statistical interaction analysis[89]. Traditional interaction analysis subtracts out main effects; the cells then only contain the interactions. Here the total effects in each cell are presented. As a result, **interactions** as used here does not conform to the precise statistical definition, but rather the informal notion of "how do things affect each other."

small model - 2 threads					small model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	0.28	1.08	0.83	0.63	none	0.21	1.10	0.70	0.55
avoid	0.51	1.10	0.83	0.78	avoid	0.47	1.14	0.69	0.72
resched.	0.55	1.12	0.91	0.83	resched.	0.55	1.07	0.74	0.76
Column	0.43	1.10	0.86	0.74	Column	0.38	1.10	0.71	0.67

medium model - 2 threads					medium model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	0.47	1.56	1.60	1.05	none	0.41	1.90	1.98	1.15
avoid	0.83	1.57	1.59	1.28	avoid	0.85	1.93	1.99	1.48
resched.	0.89	1.60	1.60	1.31	resched.	1.00	1.91	2.12	1.60
Column	0.70	1.58	1.60	1.21	Column	0.70	1.92	2.03	1.40

large model - 2 threads					large model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	0.96	2.09	2.09	1.61	none	1.20	4.32	4.26	2.81
avoid	1.44	2.11	2.08	1.85	avoid	2.23	4.31	4.32	3.46
resched.	1.48	2.12	2.11	1.88	resched.	2.65	4.54	4.61	3.81
Column	1.27	2.11	2.09	1.78	Column	1.92	4.39	4.39	3.33

Table 4.8: Effects of clustering and lock mitigation upon speedup

small model - 2 threads					small model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	1.27	1.34	1.40	1.33	none	1.29	1.53	1.58	1.46
avoid	1.46	1.34	1.37	1.39	avoid	1.67	1.55	1.52	1.58
resched.	1.52	1.37	1.52	1.47	resched.	1.91	1.50	1.60	1.66
Column	1.41	1.35	1.43	1.40	Column	1.60	1.53	1.57	1.57

medium model - 2 threads					medium model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	1.47	1.53	1.62	1.54	none	1.89	2.05	2.25	2.06
avoid	1.59	1.53	1.62	1.58	avoid	2.22	2.07	2.25	2.18
resched.	1.65	1.53	1.63	1.60	resched.	2.45	2.04	2.36	2.28
Column	1.57	1.53	1.62	1.57	Column	2.17	2.05	2.29	2.17

large model - 2 threads					large model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	1.57	1.58	1.61	1.59	none	2.26	2.28	2.37	2.30
avoid	1.66	1.58	1.60	1.62	avoid	2.44	2.26	2.36	2.35
resched.	1.66	1.58	1.61	1.62	resched.	2.71	2.34	2.50	2.51
Column	1.63	1.58	1.61	1.61	Column	2.46	2.29	2.41	2.39

Table 4.9: Effects of clustering and lock mitigation upon overlap

The general results already described can be seen from this table; both clustering and lock mitigation techniques improve simulator speeds for both two and four threads and all sizes of models. In general, larger models experience higher speedups than smaller models and more threads yield more speed as long as the models are not small. This table, however, also shows that clustering is more important – it affects the speedup more – than lock mitigation.

The presence of clustering removes most of the differences between lock mitigation heuristics. As long as some sort of clustering has been performed, performing lock reduction helps very little. This is somewhat to be expected: clustering techniques constrain the choices for the lock mitigation mechanisms and thus make them less important. This can be seen in greater detail by examining the overlap. Table 4.9 shows the overlap for the combinations of techniques, model size, and number of threads. The addition of clustering often **reduces** the overlap when lock mitigation is used. The clustering has constrained the lock mitigation techniques so that they are unable to perform their function as well. Note that this effect goes away as the models get

small model - 2 threads					small model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	4.51	1.24	1.68	2.11	none	6.06	1.39	2.27	2.68
avoid	2.87	1.23	1.64	1.79	avoid	3.55	1.37	2.20	2.20
resched.	2.76	1.22	1.67	1.78	resched.	3.45	1.40	2.17	2.19
Column	3.29	1.23	1.66	1.89	Column	4.20	1.39	2.21	2.35

medium model - 2 threads					medium model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	3.14	0.98	1.01	1.46	none	4.65	1.08	1.14	1.79
avoid	1.91	0.97	1.02	1.24	avoid	2.62	1.07	1.13	1.47
resched.	1.86	0.96	1.02	1.22	resched.	2.45	1.06	1.12	1.43
Column	2.23	0.97	1.02	1.30	Column	3.10	1.07	1.13	1.55

large model - 2 threads					large model - 4 threads				
	none	IBC	iDSC	Row		none	IBC	iDSC	Row
none	1.63	0.76	0.77	0.98	none	1.88	0.53	0.56	0.82
avoid	1.16	0.75	0.77	0.88	avoid	1.09	0.52	0.55	0.68
resched.	1.13	0.74	0.76	0.86	resched.	1.02	0.51	0.54	0.66
Column	1.29	0.75	0.77	0.90	Column	1.28	0.52	0.55	0.72

Table 4.10: Effects of clustering and lock mitigation upon dilation

larger and as the number of threads increase. This is probably because as the models get larger the number of tasks increases and the less likely an attempt by two conflicting tasks to execute concurrently becomes. The effect is also much less pronounced for iDSC, which tries to retain some notion of the critical path while clustering, than IBC, which ignores the critical path.

Returning to Table 4.8, the use of lock reduction techniques appears to flatten the differences between clustering techniques in an unusual way. The difference between the two clustering techniques is not greatly affected, but the difference between no clustering and using either technique is reduced. This is simply because the lock reduction techniques are achieving part of the effects of clustering. This can be seen by looking at Table 4.10, which is a similar table for dilation. Once clustering is performed, dilation is mostly unchanged as lock mitigation is performed. However, lock mitigation by itself always noticeably reduces dilation.

Small models gain the most benefit from clustering techniques, more than doubling their performance when IBC is used. Yet this is still insufficient to achieve large speedups. Why are small models so hard? It is because they are typically tightly interconnected; there are not very many large, independent subsystems. This has two effects: first, more data must be shared between the threads. Second, creating schedules without large amounts of waiting for that data to cross the threads is difficult. Variability in codeblock execution times, makes predicting the task costs and scheduling appropriately even more difficult. Further improvements may require means to decompose codeblocks, which is not possible when they are treated as black boxes.

The medium and large models with four threads behave differently from the small models. The use of iDSC with lock rescheduling produces significantly better results than all other combinations. This is because the overlap is significantly greater while the dilation is not. Note that for two threads, this combination works as well as any other.

In conclusion, parallelization should use both iDSC and lock rescheduling when the model is of medium or large size, particularly when there are four or more threads. For small models, IBC should be used, but users should be aware that the additional use of lock mitigation will have only small effects.

4.6.5 Odd Thread Counts

Results for two and four threads have been presented so far. One could argue that for the medium and large models parallelization for two or four threads is very simple and obvious, as there are a power-of-two number of tiles in the CMP models. The argument would then claim that parallelization by hand would be possible and automatic techniques would be unnecessary. To counter this argument, this subsection presents the results of three-threaded simulation of the 8-tile and larger CMP models. Load balancing between three threads for these models requires that tiles be split between threads. Performing this partitioning by hand would be awkward and time-consuming.

Figure 4.22 shows the speedup for two, three, and four-threaded simulators built for the 4-, 8-, 16-, and 32-way CMP models using lock rescheduling and iDSC. The three-threaded simulator performs well, achieving speedup between that of two and four threads, including the now-expected superlinear speedup for the ***CMP16*** and ***CMP32*** models.

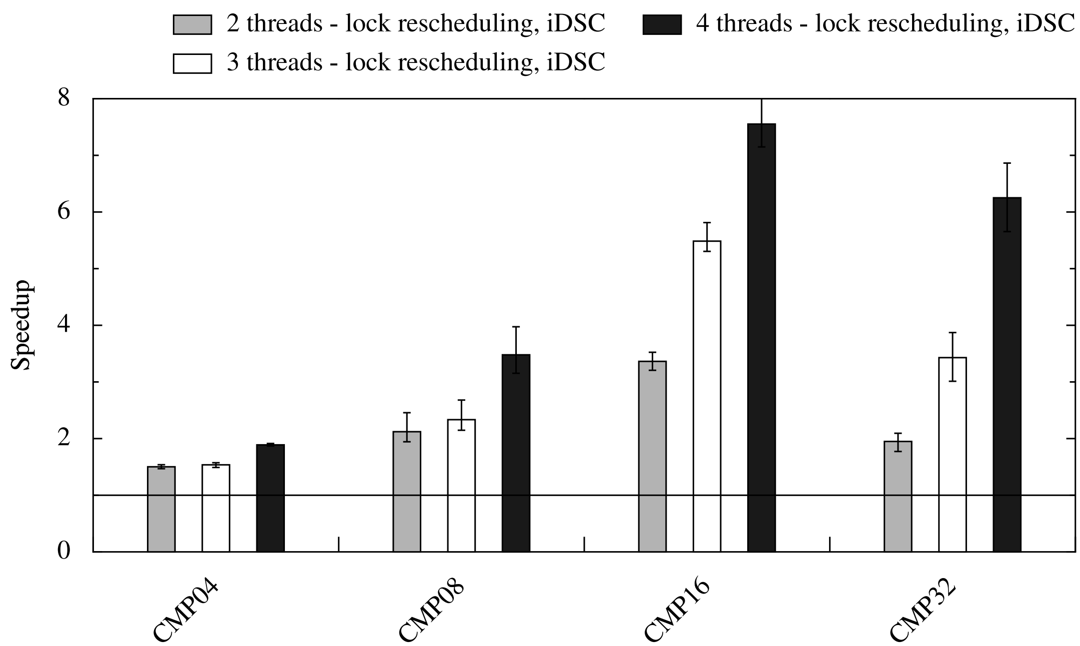


Figure 4.22: Comparison of speedup with different thread counts

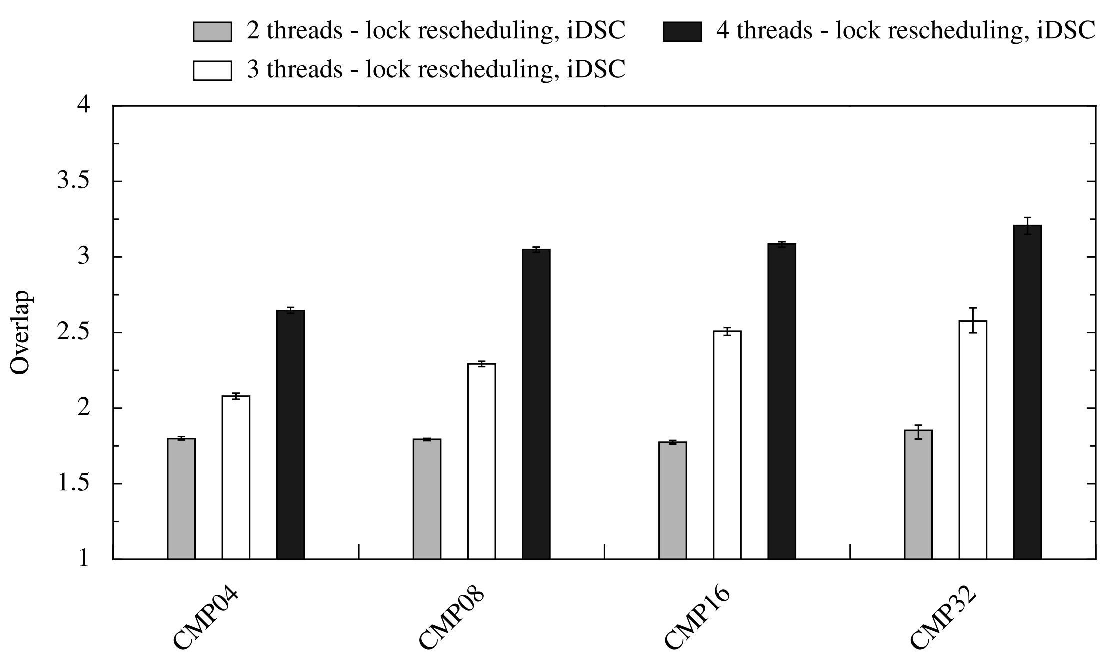


Figure 4.23: Effects of thread count on overlap

Figure 4.23 shows the effects of an odd thread count on the overlap. The speedup due to work overlap increases smoothly with the number of threads, indicating that the scheduler is successfully splitting the tiles. This is further indicated by the load imbalance measurements shown in Table 4.11. For the 8, 16, and 32-way models load balancing is nearly perfect; for the ***CMP04*** model there is a small amount of imbalance with three threads, but performance still scales properly.

	2 threads	3 threads	4 threads
CMP04	0.06	12.70	1.01
CMP08	0.03	0.00	0.03
CMP16	0.01	0.01	0.01
CMP32	0.01	0.00	0.01

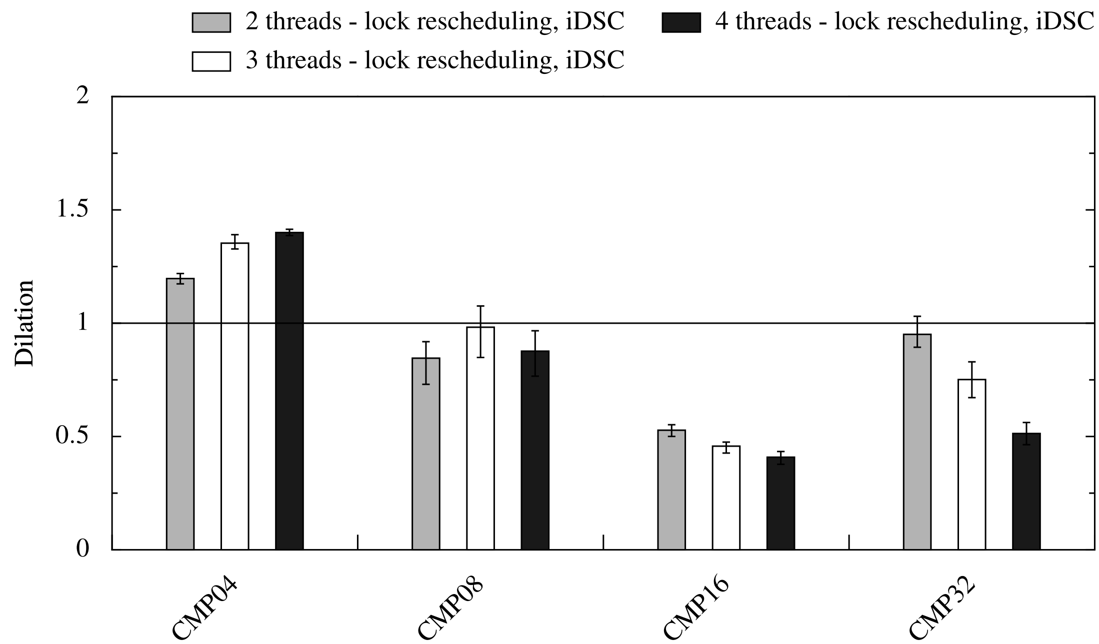
Table 4.11: Effects of thread count on percent load imbalance

Figure 4.24 shows the effects of the odd thread count on the dilation. The dilation shows interesting behavior for the ***CMP08*** model, becoming more when three threads are used than for two or four. The working set of this model's uniprocessor simulator is larger than a processor's cache, but when divided properly among two threads, it fits well within each cache. However, with three or four threads, more sharing misses occur but there are not corresponding reductions in capacity misses. Four threads do not have more misses than three threads in this case because the splitting of tiles necessary with three threads increases the amount of data which must be shared. This model highlights the desirability of future work which investigates the tuning of these heuristics and finds when the number of threads used is not optimal.

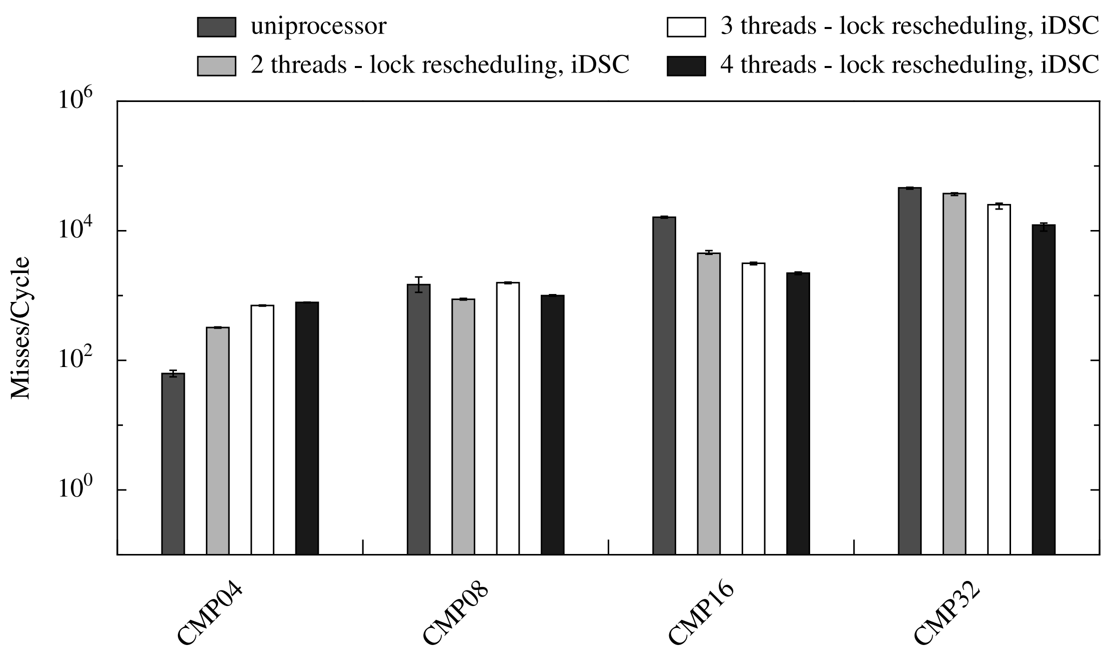
4.6.6 Summary: Traditional Multiprocessors

For a traditional multiprocessor, the following has been shown:

- Both lock avoidance and lock rescheduling are very effective at reducing the number of locking operations and the amount of time spent waiting at locks. Either heuristic is effective at improving parallel simulation performance.
- Lock rescheduling is more effective than lock avoidance when there are four threads.



(a) Dilation



(b) Cache misses

Figure 4.24: Effects of thread count on dilation

- Clustering techniques in combination with lock rescheduling are effective at improving cache locality and reducing dilation. These techniques can provide significant speedups for most models.
- Larger models exploit additional threads beyond two more effectively than smaller models.
- There exists a "sweet spot" of model size versus the number of processors where the reduction in capacity misses less the increase in sharing misses is maximized.
- Of the clustering techniques, Instance-Aware Dominant Sequence Clustering is better for smaller models or for more threads. Instance-Based Clustering is better for large models if few threads are used.

In conclusion, parallelization should use both iDSC and lock rescheduling when the model is of medium or large size, particularly when there are four or more threads. For small models, IBC should be used, but users should be aware that the additional use of lock mitigation will have only small effects.

Because there is a "sweet spot" of model size versus the number of processors, it would be desirable to a priori estimate the optimal number of processors to use for a given model. Unfortunately, the results in this section give little guidance. Certainly larger models will have larger working sets and be better able to use more processors, but the definition of model size needs to be further refined beyond just numbers of signals or instances or codeblocks to include information about the granularity of codeblocks. Furthermore, as was seen with the ***I2-CMP*** model, the use of more processors may lead to larger load imbalance. A possible procedure would be to use a size metric to form an initial estimate of the number of processors and then to progressively reduce the number of processors if load imbalance is an issue.

4.7 Chip Multiprocessor Evaluation

A fairly recent development is the wide availability of simple chip multiprocessors (CMPs). A chip multiprocessor is a microprocessor with more than one processor core on a single die. These cores may or may not share portions of the cache hierarchy. This section presents a preliminary evaluation of the multiprocessor scheduling heuristics introduced in the last section upon a two-way chip multiprocessor. As in the previous section, the section first shows the effects of lock mitigation, then the effects of clustering, and finally interactions between the two.

4.7.1 Methodology

The evaluation system is a single processor system with one Intel®Pentium® D running at 2.8 GHz. This system has 1 megabytes of L2 cache per core and 4 gigabytes of memory. The L2 cache is not shared; cache-to-cache data transfers need not go through memory, but do require access to the system bus. The system runs Fedora Core release 3, with kernel version 2.6.15.4. All simulators are compiled using gcc 3.4.2 with the default compilation flags provided by LSE's `ls-build` script.

The same nine models are used for the evaluation, but the parameters of the iDSC heuristic have been changed to match the new expected communication costs. The cost in cycles of inter-task communication from task i to j running in different threads is now taken to be: $15*(1+NS_{ij})$ where NS_{ij} is the number of signals generated by i and used by j; 15 cycles is approximately the cost of an access to an L2 cache. This figure is meant to reflect the lower cost for cache-to-cache transfers in a chip multiprocessor, but is much too low for this system architecture.

4.7.2 Lock Mitigation

This subsection evaluates the effectiveness of the lock mitigation heuristics at reducing waiting for locks. Figure 4.25 shows the average speedup for each model's simulator when parallelized onto two threads using no lock mitigation, lock avoidance, and lock rescheduling. Speedups are measured relative to the uniprocessor simulator for each model. As in the previous section, the error bars show the highest and lowest speedups for each model.

Both lock avoidance and rescheduling improve performance, with lock rescheduling the more effective of the two. This is precisely what was seen for the traditional multiprocessor (Figure 4.10). However the size of the effect is smaller for the larger models such as ***CMP16*** and ***CMP32***.

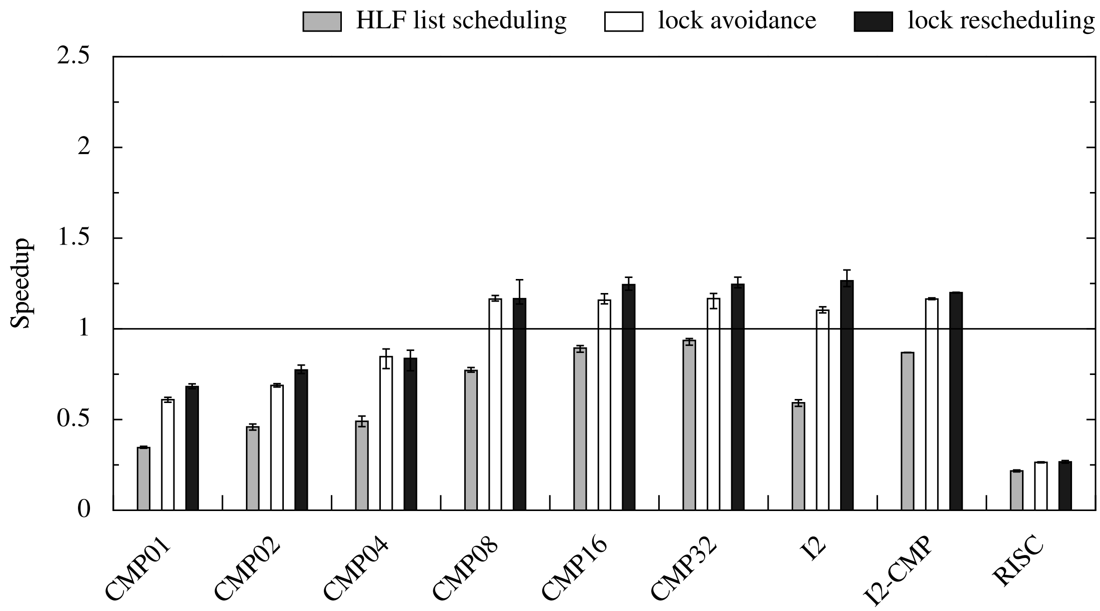
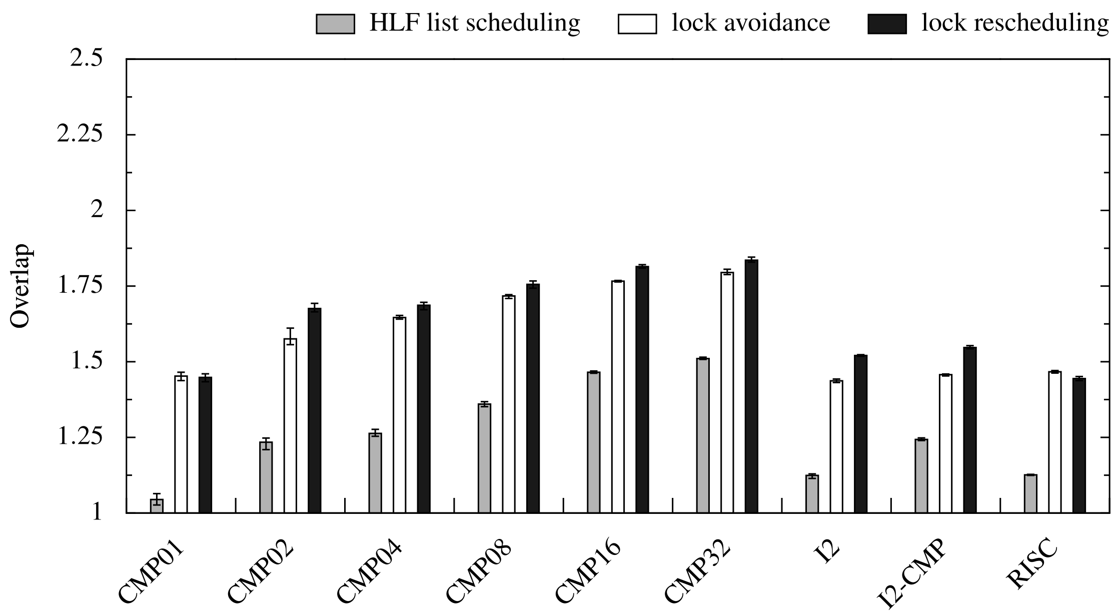


Figure 4.25: Speedup for two-threaded CMP parallelization with lock mitigation

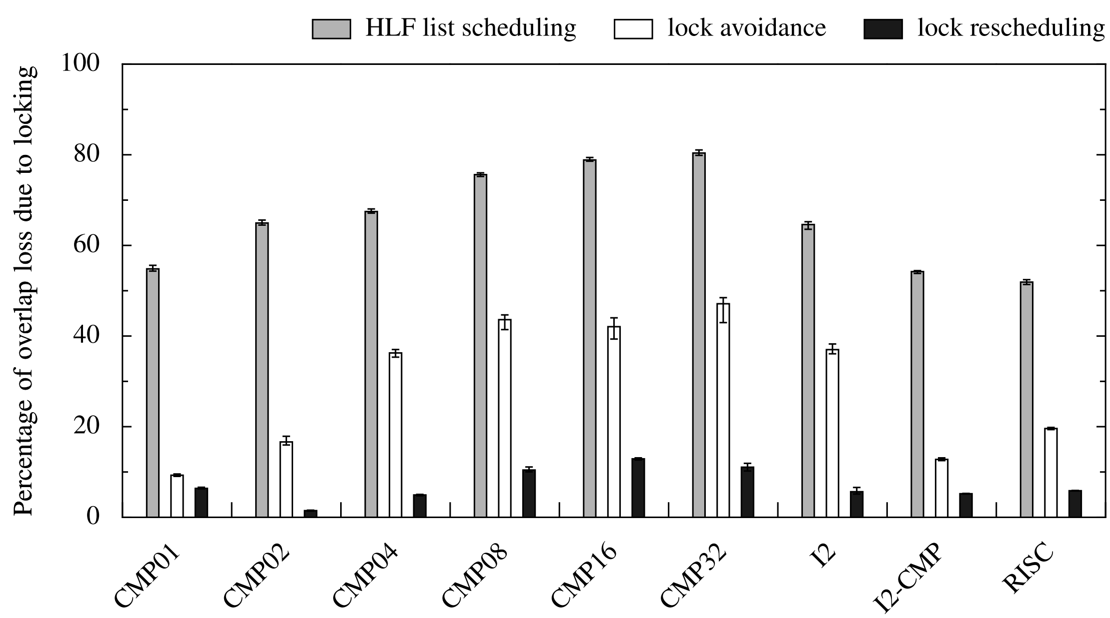
A loss of effectiveness in lock mitigation is not the reason for the reduction in effect size for the largest models. Figure 4.26 shows the effect of lock mitigation on overlap; the amount of overlap is very similar to what was seen for the traditional multiprocessor (Figure 4.11). Indeed, reduction in the percentage of overlap loss due to locking is even more impressive because it starts from a higher number in the chip multiprocessor case.

The reason for the loss of effect size for larger models is to be found in the effects of lock mitigation upon cache locality. Recall that there was a serendipitous improvement in cache locality for the traditional multiprocessor as lock mitigation tended to place tasks calling codeblocks from the same instance in the same thread. Figure 4.27 shows that this effect is still present for the CMP, but is not as strong as it was for the traditional multiprocessor (Figure 4.12(a)), except for the ***I2*** model.

To summarize, lock mitigation performs much the same on a chip multiprocessor as it did on a traditional multiprocessor. Either lock avoidance or lock rescheduling are effective, and lock rescheduling is generally more effective.



(a) Overlap



(b) Percent of lost overlap due to locks

Figure 4.26: Effects of lock mitigation on overlap for two CMP threads

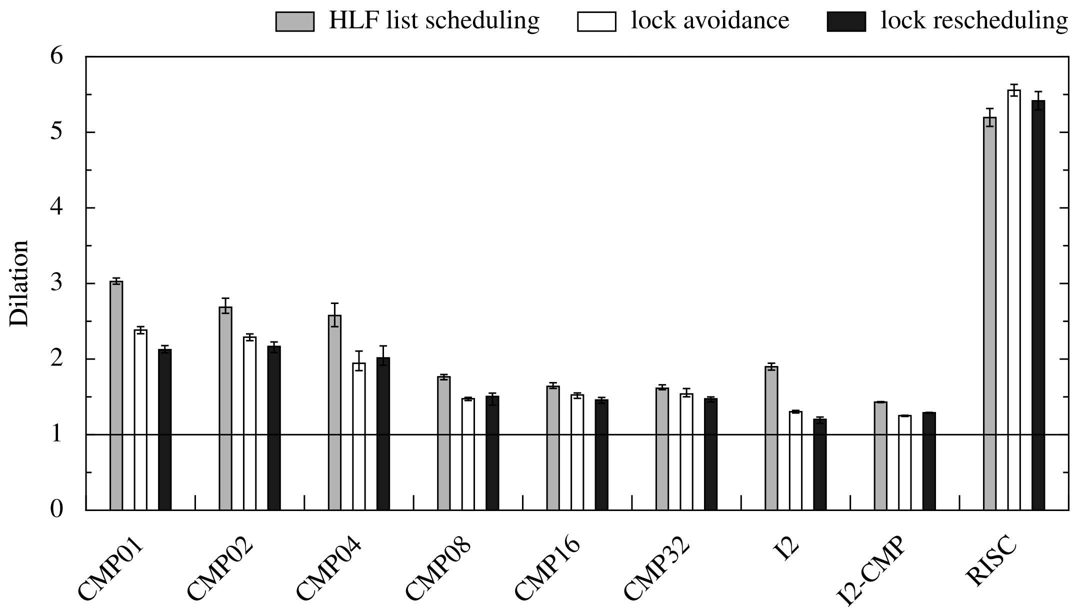


Figure 4.27: Effects of lock mitigation on dilation for two CMP threads

4.7.3 Clustering

This subsection presents the effectiveness of clustering at improving cache locality. Because the caches are not shared in the system used for this evaluation, clustering should improve locality much as it did for the traditional multiprocessor. However the effect on dilation and speedup should be less than it was for the traditional multiprocessor, as the latency of cache-to-cache transfers should be reduced.

Figure 4.28 shows the average speedup for each model's simulator when parallelized onto two threads using no clustering, Instance-Based Clustering, and Instance-Aware Dominant Sequence Clustering. Speedups are measured relative to the uniprocessor simulator for each model. As always, the error bars show the highest and lowest for speedups for each benchmark. After clustering, lock rescheduling is used in each case with processor assignments constrained by the clustering.

Both clustering techniques improve simulator performance for all the models. Superlinear speedup is possible for the ***CMP08*** model. This is different from the results obtained using a traditional multiprocessor (Figure 4.16) in that the maximum speedup occurs with a smaller model. This shift in the size at which maximum speedup occurs is surprising as the per-core cache

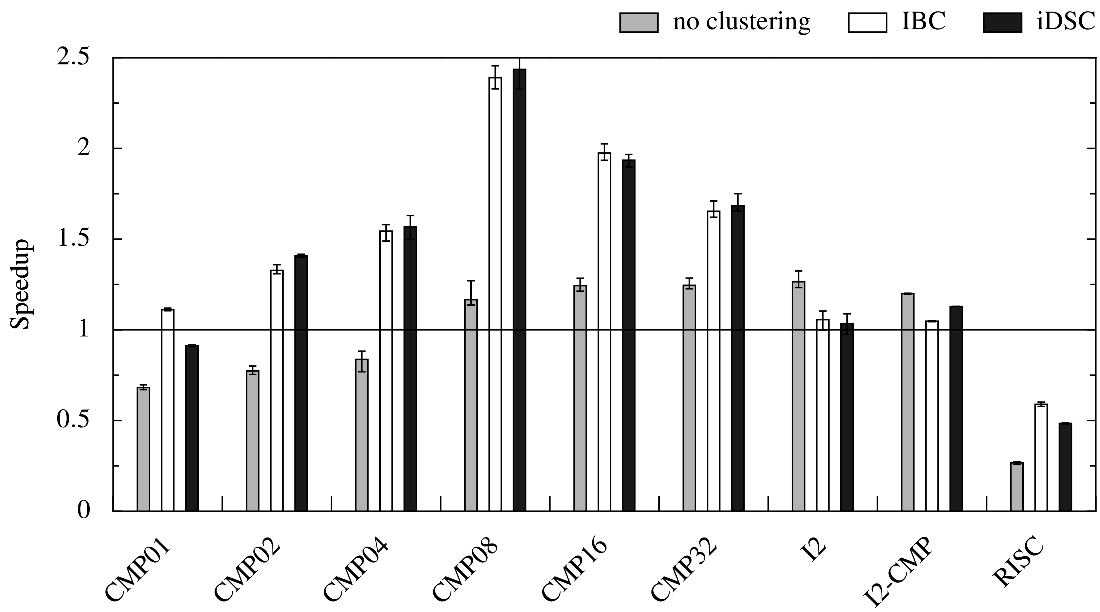


Figure 4.28: Speedup for two-threaded CMP parallelization with clustering

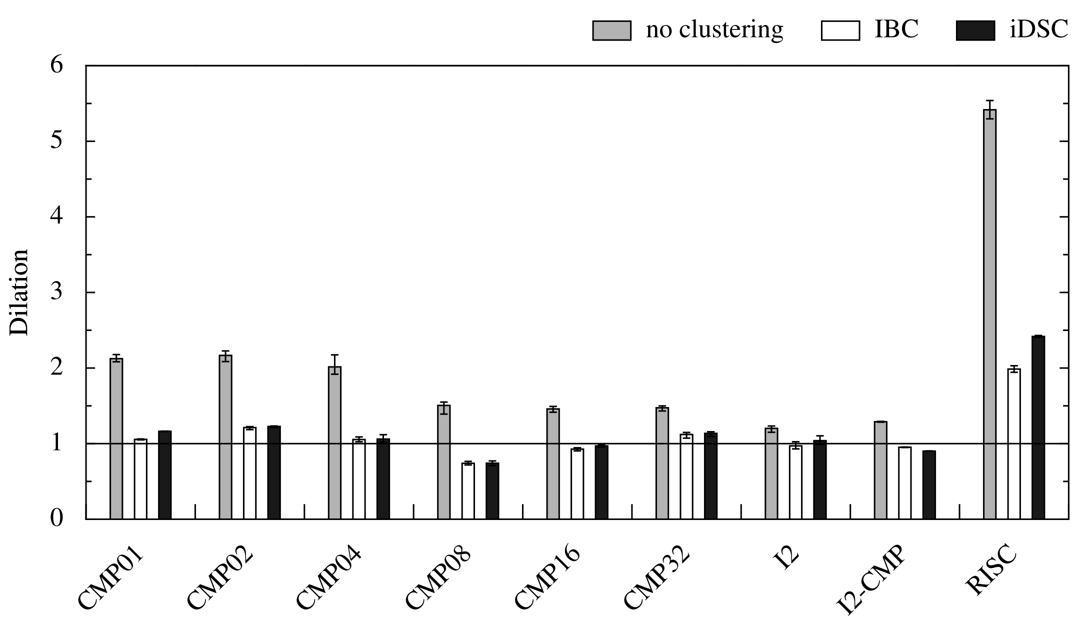


Figure 4.29: Effect of clustering on dilation for two CMP threads

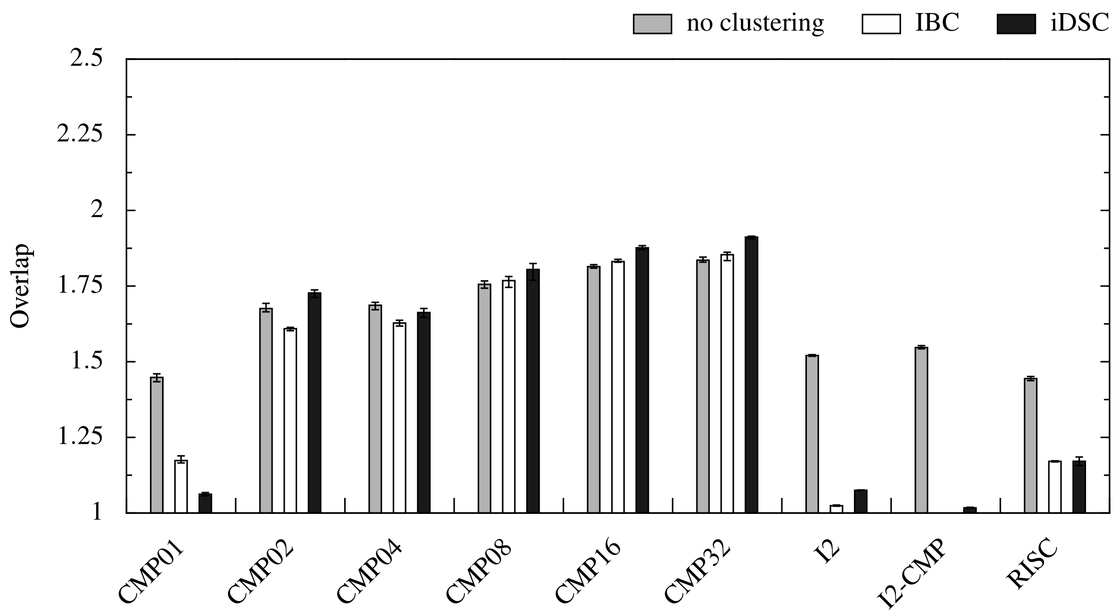
sizes are the same in this CMP as in the traditional multiprocessor which was used. Figure 4.29 shows the effects of clustering on dilation. Clustering reduces dilation for all models as it did on a traditional multiprocessor, though not by as much; this does agree with intuition.

Unfortunately, the impact of clustering upon cache misses cannot be directly measured. The performance counters of the Pentium® D do not count L2 cache misses directly, though they will count memory accesses. However the granularity and accuracy available appear insufficient to use these counters with any confidence. For example, the counters report that the uniprocessor ***CMP32*** simulator requires fewer than one memory access per simulated cycle for the 1-megabyte cache of the processor. This is inconceivable in light of the last section's results, where a 1-megabyte cache resulted in about three orders of magnitude more accesses to memory. The sampling granularity delivered by the counter may not be that which was requested and at the granularity delivered, no effects upon the number of memory accesses are seen.

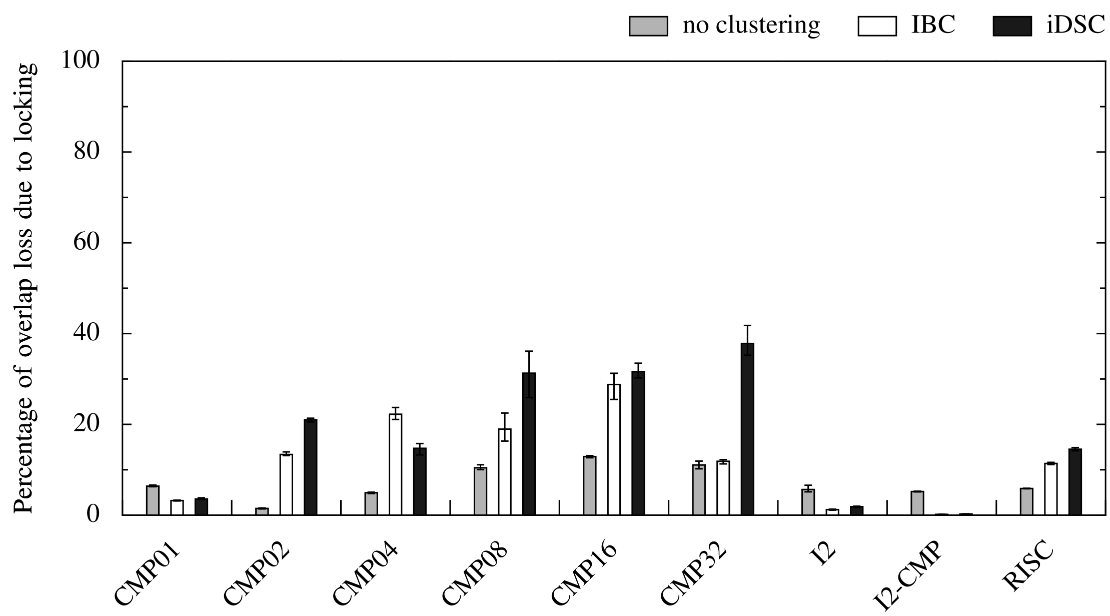
Figure 4.30(a) shows effects of clustering upon overlap. For the smallest models (***CMP01*** and ***RISC***), clustering reduces overlap severely. The ***CMP01***, ***I2***, ***I2-CMP***, and ***RISC*** models experience load imbalance, as seen in Table 4.12. For the largest models, clustering has little effect on overlap. This is similar to the results for the traditional multiprocessor (Table 4.18(a)), except that when iDSC improves overlap over no clustering, it does not do so as strongly. This is to be expected; inter-thread communication costs should affect overlap less when they are smaller, as they are for a chip multiprocessor. Note that just as in the traditional multiprocessor, clustering constrains the lock rescheduling and leads to a larger percentage of the wait time being spent in locks.

	none	IBC	iDSC
CMP01	0.22	60.90	84.27
CMP02	0.11	1.01	0.11
CMP04	0.06	0.73	0.06
CMP08	0.03	8.69	0.03
CMP16	0.01	6.74	0.01
CMP32	0.01	7.52	0.01
I2	0.11	94.37	81.90
I2-CMP	0.70	95.75	90.63
RISC	3.37	55.06	1.12

Table 4.12: Percent load imbalance for two-threaded CMP parallelization with clustering



(a) Overlap



(b) Percent of overlap loss due to locks

Figure 4.30: Effects of clustering on overlap for two CMP threads

small model - 2 threads				
	none	IBC	iDSC	Row
none	0.37	1.08	0.95	0.72
avoid	0.57	1.08	0.94	0.83
resched.	0.63	1.10	1.00	0.88
Column	0.51	1.08	0.96	0.81

medium model - 2 threads				
	none	IBC	iDSC	Row
none	0.61	1.67	1.72	1.20
avoid	1.01	1.64	1.70	1.41
resched.	1.04	1.70	1.72	1.45
Column	0.86	1.67	1.71	1.35

large model - 2 threads				
	none	IBC	iDSC	Row
none	0.91	1.60	1.60	1.32
avoid	1.16	1.60	1.60	1.44
resched.	1.24	1.62	1.64	1.49
Column	1.09	1.61	1.62	1.42

Table 4.13: Effects of clustering and lock mitigation upon speedup for two CMP threads

Overall, clustering leads to performance improvement for all the models except ***I2*** and ***I2-CMP***. The combination of lock rescheduling with clustering makes parallelization beneficial for all the CMP family of models, though the benefit is less for the ***CMP16*** and ***CMP32*** models than it was on a traditional multiprocessor. This reduction in benefit for these models, combined with the increase in benefit for the ***CMP08*** model is interesting, especially in light of the fact that per-processor core cache size has not changed. This may be an indication of the insufficiency of the task cost estimation method or the communication cost estimation and warrants attention in future work.

4.7.4 Interactions

Table 4.13 shows the average speedup of each combination of lock mitigation and clustering heuristics for the different sizes of models. The heuristics interact much as they did for the traditional multiprocessor system: the effect of clustering is much larger and once clustering is done, the uses of lock reduction techniques makes only a minor difference. As before, the benefit comes mainly from improved cache locality.

As with traditional multiprocessor systems, the preferred combination of heuristics would be lock rescheduling plus iDSC for medium and large models and lock rescheduling plus IBC for small models.

4.7.5 Summary: Chip Multiprocessors

For a chip multiprocessor, the following has been shown:

- The effectiveness of lock mitigation on a chip multiprocessor is similar to that on a traditional multiprocessor. Either lock avoidance or lock rescheduling improve performance, with lock rescheduling resulting in slightly higher performance.
- Either clustering technique works well, but Instance-Aware Dominant Sequence Clustering is better for medium and large models.
- Maximum speedup due to clustering takes place for smaller models than for traditional multiprocessors; this result is surprising.

In conclusion, the preferred combination of heuristics would be lock rescheduling plus iDSC for medium and large models and lock rescheduling plus IBC for small models.

4.8 Simultaneous Multithreading Evaluation

Another common microarchitectural technique is multithreading. A simultaneous multithreading processor maintains state for multiple threads of execution and may issue instructions from multiple threads in the same clock cycle. This section presents a preliminary evaluation of the multiprocessor scheduling heuristics introduced in this chapter upon a two-way simultaneous multithreading processor. The section first shows the effects of lock mitigation, then the effects of clustering, and finally interactions between the two.

4.8.1 Methodology

The evaluation system is a single processor system with one Intel®Pentium®4 Processor with Hyperthreading running at 3.0 GHz. This processor supports two simultaneous threads and 512 KB of L2 cache; the system has 2 gigabytes of memory. All caches and the processor datapath are shared. The system runs Fedora Core release 3, with kernel version 2.6.12-1.1381_FC3smp. All simulators are compiled using gcc 3.4.4 with the default compilation flags provided by LSE's `ls-build` script.

The same nine models are used for the evaluation, but the parameters of the iDSC heuristic have been changed to match the new expected communication costs. The cost in cycles of inter-task communication from task i to j running in different threads is now taken to be: $5*(1+NS_{ij})$ where NS_{ij} is the number of signals generated by i and used by j; 5 cycles is meant to reflect an access to the shared L1 cache, though this number is perhaps large.

4.8.2 Lock Mitigation

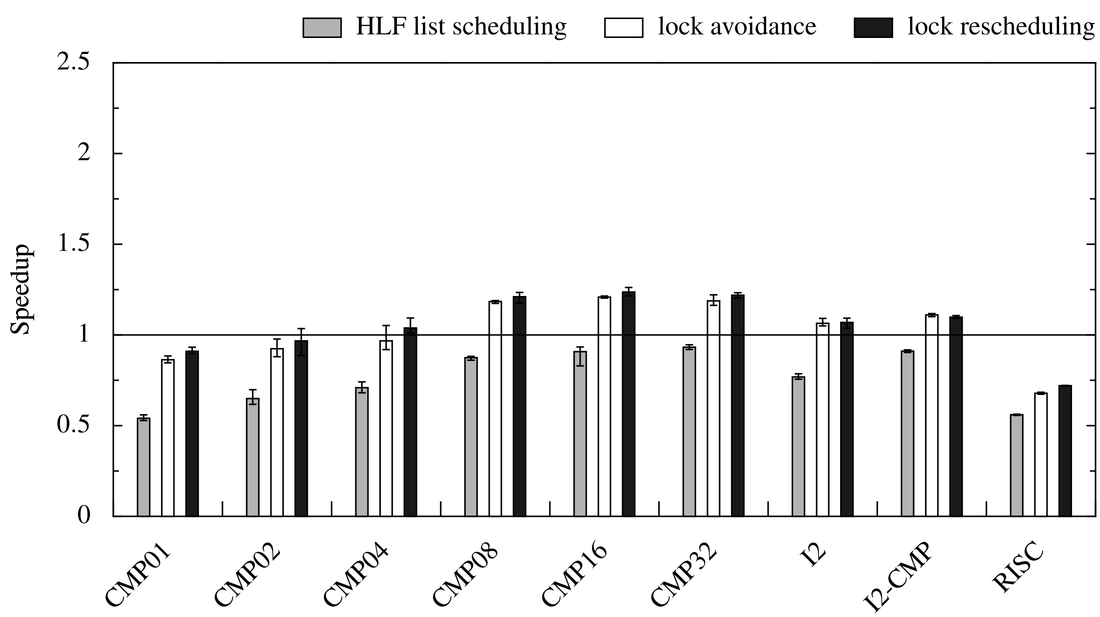


Figure 4.31: Speedup for two-threaded SMT parallelization with lock mitigation

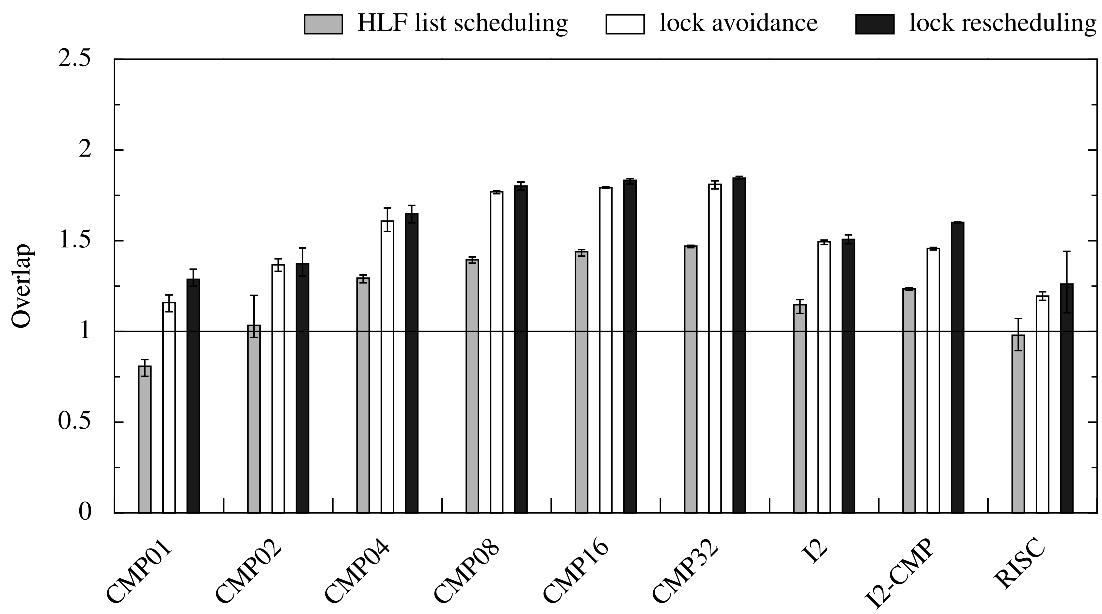
This subsection evaluates the effectiveness of the lock mitigation heuristics at reducing waiting for locks. Figure 4.31 shows the average speedup for each model's simulator when parallelized onto two threads using no lock mitigation, lock avoidance, and lock rescheduling. Speedups are measured relative to the uniprocessor simulator for each model. As in the previous section, the error bars show the highest and lowest speedups for each model.

The base HLF list scheduling performs much better for smaller models than it has when parallelizing for other host systems, though it still does not achieve speedup. Once lock mitigation is added, all but the three smallest models achieve some speedup.

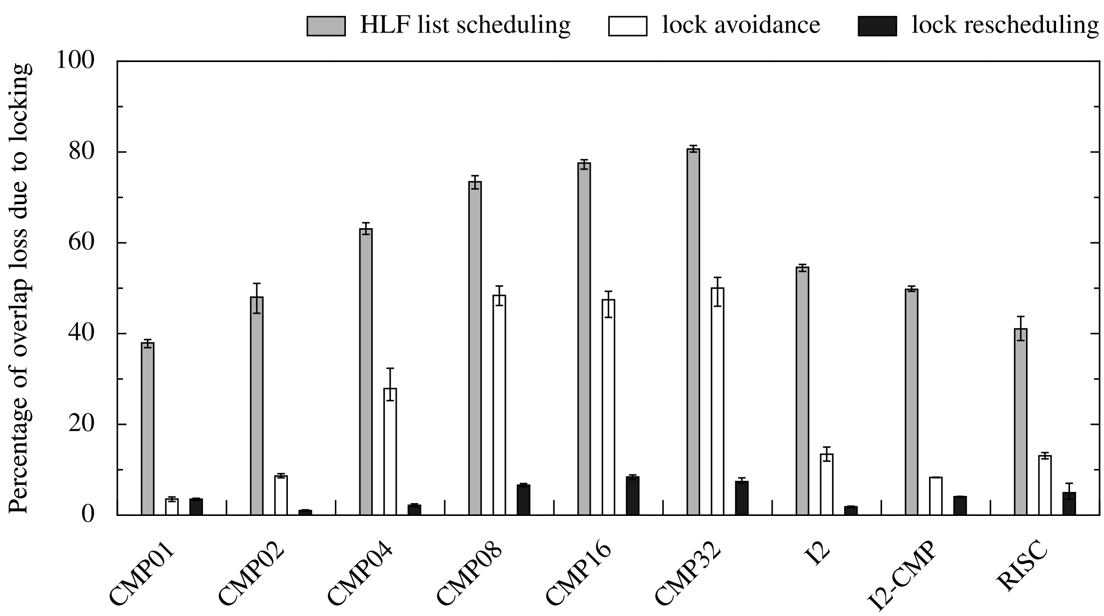
As has been seen for the other host systems, both lock avoidance and rescheduling improve performance, with lock rescheduling the more effective of the two. The difference between the two heuristics is very small. Figure 4.32 shows the effects of lock mitigation on overlap. Lock mitigation continues to improve overlap and reduce the fraction of overlap lost waiting for locks, in much the same manner as it has for the other host systems.

Lock mitigation should no longer have the side effect of improving cache behavior as the entire cache hierarchy is now shared between the threads. Figure 4.33 shows the effect of lock mitigation upon dilation; lock mitigation has some small effect, which is usually a reduction in dilation. More interesting still, dilation is still greater than one even with presumably no cache effects. Some of this is due to overhead: the implementation requires that task invocations carry the thread ID as a parameter, for example. However most of it is likely to be due to competition between the threads for pipeline resources such as the reorder buffer or the load/store queue; lock mitigation might reduce this contention. This seems even more likely in light of the fact that the atomic memory operations used to implement locks could very well serialize memory accesses for both threads in the load/store queue.

To summarize, lock mitigation for a simultaneous multithreading processor continues to increase overlap in the same fashion that it does for a traditional multiprocessor. There are also some small reductions in dilation for some models. The net result is that parallelization becomes effective for many models. Either lock avoidance or lock rescheduling can be used, but neither is preferred.



(a) Overlap



(b) Percent of lost overlap due to locks

Figure 4.32: Effects of lock mitigation on overlap for two SMT threads

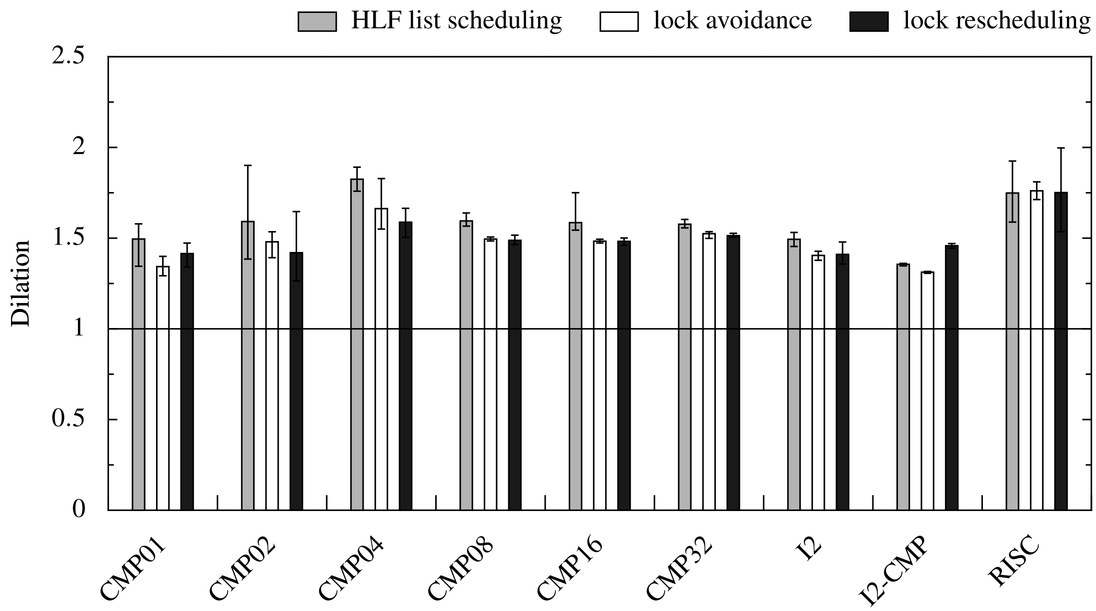


Figure 4.33: Effects of lock mitigation on dilation for two SMT threads

4.8.3 Clustering

This subsection presents the effectiveness of clustering at improving cache locality. Because the entire cache hierarchy is shared, clustering should not improve locality and one expects to see little performance difference from clustering unless it interferes with lock mitigation.

Figure 4.34 shows the average speedup for each model's simulator when parallelized onto two threads using no clustering, Instance-Based Clustering, and Instance-Aware Dominant Sequence Clustering. Speedups are measured relative to the uniprocessor simulator for each model. As always, the error bars show the highest and lowest for speedups for each benchmark. After clustering, lock rescheduling is used in each case with processor assignments constrained by the clustering.

Clustering achieves slight performance gains for about half of the models, but reduces performance for the other models. This is very different from the results for the traditional multiprocessor, and it occurs because the cache hierarchy is totally shared. Figure 4.35 shows the effects on dilation; clustering has very little effect except for the ***I2*** and ***I2-CMP*** models, where as can be seen in Table 4.14, clustering can create enormous load imbalance and virtually "unparallelize" the simulator. The effect of clustering on cache misses cannot be measured reliably

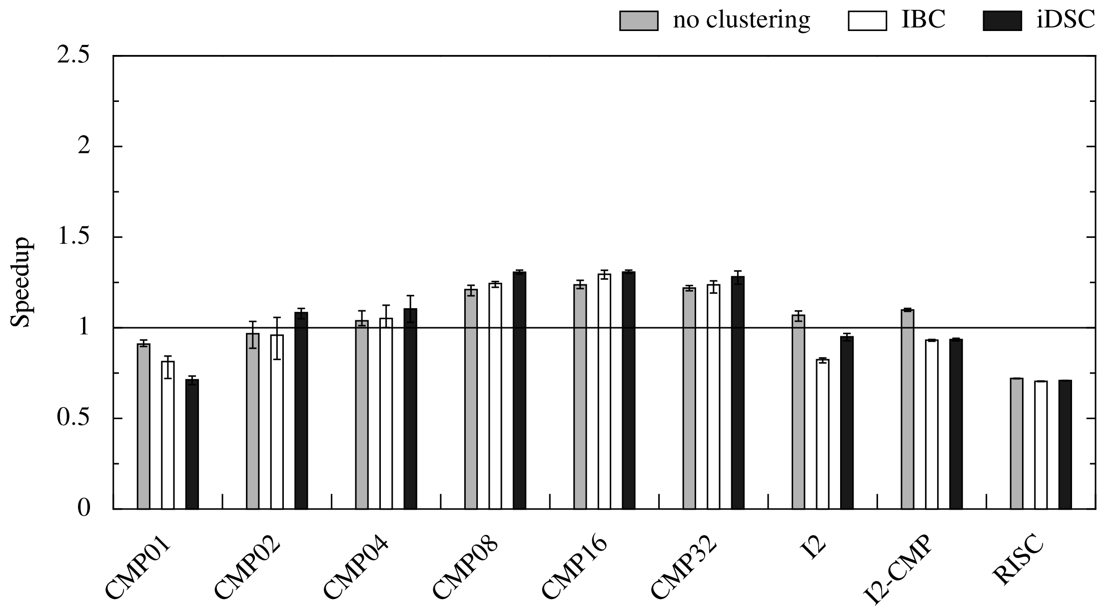


Figure 4.34: Speedup for two-threaded SMT parallelization with clustering

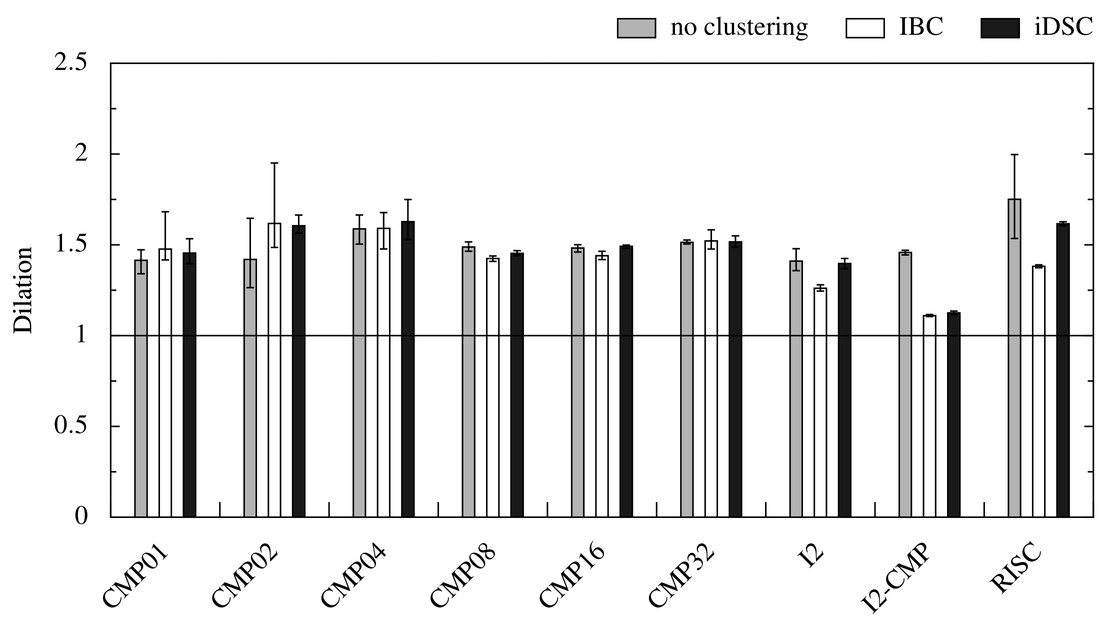


Figure 4.35: Effect of clustering on dilation for two SMT threads

because the performance counters of the Pentium® 4 processor have the same issues as those of the Pentium® D.

Figure 4.36 shows the effects of clustering on overlap. The ambiguous results from the traditional multiprocessor appear here as well: clustering improves overlap for some models and degrades it for others. The iDSC heuristic outperforms IBC for all but the ***CMP01*** model. The ***CMP01***, ***I2***, ***I2-CMP***, and ***RISC*** models suffer from load imbalance with clustering as seen in Table 4.14 and have less overlap than without clustering; however note that all of these models do achieve some overlap.

	none	IBC	iDSC
CMP01	0.22	60.90	84.27
CMP02	0.11	1.01	0.11
CMP04	0.06	0.73	0.17
CMP08	0.03	8.69	0.03
CMP16	0.01	6.74	0.01
CMP32	0.01	7.52	0.01
I2	0.11	94.37	3.42
I2-CMP	0.70	95.75	93.59
RISC	3.37	55.06	1.12

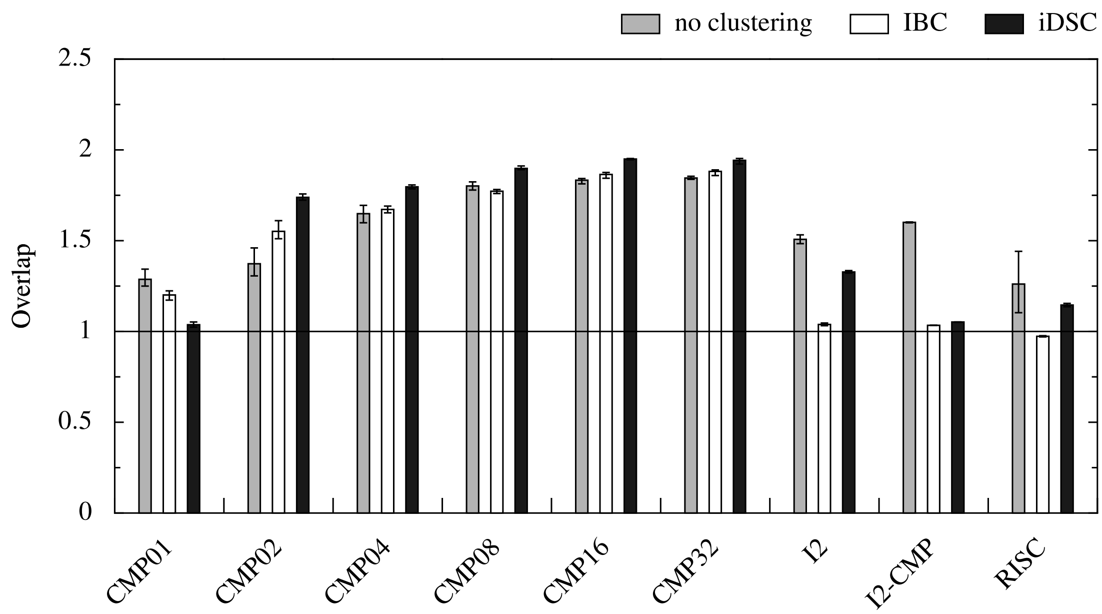
Table 4.14: Percent load imbalance for two-threaded SMT parallelization with clustering

Overall, clustering can lead to performance improvement, though for many models it may lead to performance degradation. Parallelization is effective, though not overwhelmingly so, for about half of the models; the maximum speedup achieved is 1.34.

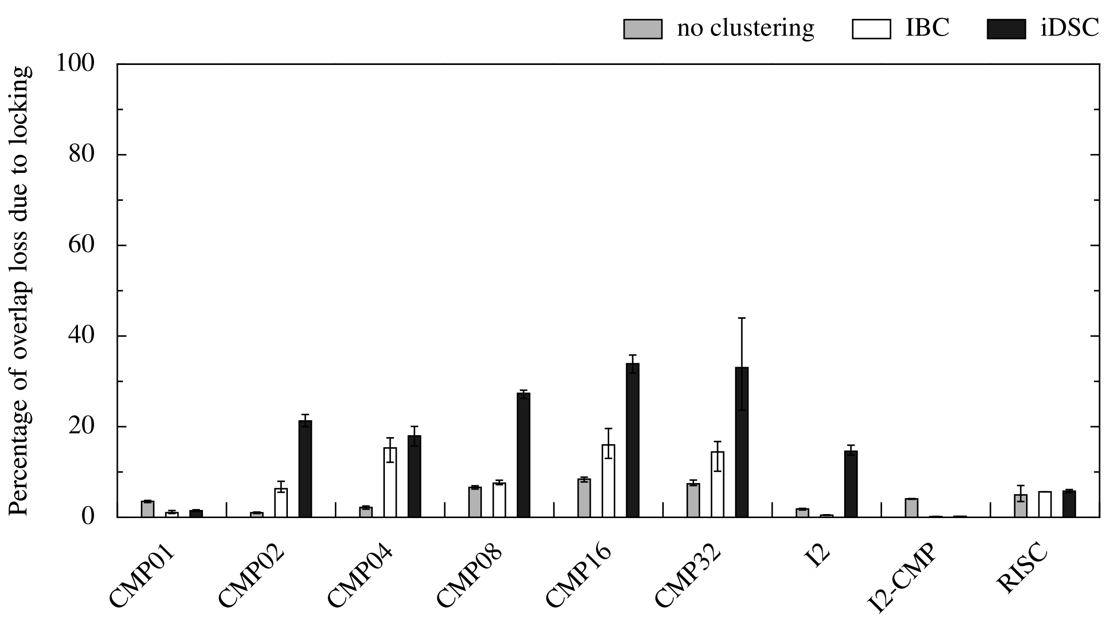
4.8.4 Interactions

Table 4.15 shows the average speedup of each combination of lock mitigation and clustering techniques for the different sizes of models. The heuristics interact differently in simultaneous multithreading processors than they do for the traditional multiprocessor or chip multiprocessor; clustering is now not always the larger effect. Yet as before, once clustering is done, lock mitigation techniques still make only a minor difference. This is because clustering has constrained lock mitigation.

As with other models, IBC performs marginally better than iDSC for small models and marginally to slightly worse for medium and large models. This is surprising at first glance;



(a) Overlap



(b) Percent of overlap loss due to locks

Figure 4.36: Effects of clustering on overlap for two SMT threads

small model - 2 threads				
	none	IBC	iDSC	Row
none	0.59	0.88	0.85	0.76
avoid	0.86	0.88	0.84	0.86
resched.	0.90	0.86	0.85	0.87
Column	0.77	0.87	0.85	0.83

medium model - 2 threads				
	none	IBC	iDSC	Row
none	0.78	1.09	1.18	1.00
avoid	1.07	1.08	1.18	1.11
resched.	1.11	1.07	1.15	1.11
Column	0.98	1.08	1.17	1.07

large model - 2 threads				
	none	IBC	iDSC	Row
none	0.92	1.19	1.22	1.10
avoid	1.18	1.19	1.22	1.20
resched.	1.20	1.19	1.21	1.20
Column	1.09	1.19	1.21	1.16

Table 4.15: Effects of clustering and lock mitigation upon total speedup for two SMT threads

when the cache hierarchy is totally shared the size of the model should not matter. The reason is that these results are skewed in each case by a model which becomes load unbalanced by clustering: ***CMP01*** in the case of the small models and ***I2-CMP*** in the case of the large models. This imbalance has occurred to some extent for *all* of the host systems; understanding the imbalance is a worthy goal for future work.

As with traditional multiprocessor and chip multiprocessor systems, the preferred combination of techniques would be lock rescheduling plus iDSC for medium and large models. However, unlike those systems, lock rescheduling without any clustering should be used for small models, if parallelization is performed at all.

4.8.5 Summary: Simultaneous Multithreading Processors

For a simultaneous multithreading processor, the following has been shown:

- The effectiveness of lock mitigation on a simultaneous multithreading processor is similar to that on a traditional multiprocessor. Either lock avoidance or lock rescheduling may be used, but neither is to be preferred.

- Clustering leads to only marginal performance improvements and may degrade performance by creating load imbalances, particularly in small models.

In conclusion, the preferred combination of techniques would be lock rescheduling plus iDSC for medium and large models; foregoing iDSC would also be acceptable. Lock rescheduling without any clustering should be used for small models, if parallelization is performed at all.

4.9 Summary: Scheduling for Parallel Structural Simulation

This chapter has shown that structural simulators using the Heterogeneous Synchronous Reactive model of computation can be automatically parallelized and that the method used for scheduling can have a large impact upon simulation speed. A simple list scheduling approach results in parallel simulators which are slower than the corresponding uniprocessor simulators.

The parallel simulator scheduling problem is an instance of the multiprocessor task scheduling problem with precedence constraints, sequence dependence, resource constraints, and com-

Model Size	Architecture			
	Traditional Multiprocessor		Chip Multiprocessor	Simultaneous Multi-threaded
	2 threads	4 threads	2 threads	2 threads
Small	Lock rescheduling + IBC	Lock rescheduling + IBC	Lock rescheduling + IBC	Lock rescheduling
Medium	Lock rescheduling + iDSC	Lock Rescheduling + iDSC	Lock rescheduling + iDSC	Lock rescheduling + iDSC
Large	Lock Rescheduling + iDSC	Lock Rescheduling + iDSC	Lock rescheduling + iDSC	Lock rescheduling + iDSC

Table 4.16: Recommended scheduling heuristics

munication costs. I have introduced two new lock mitigation heuristics to reduce time spent waiting for locks to meet resource constraints and two new clustering heuristics to improve cache locality and reduce the effects of sequence dependence. These heuristics have been evaluated for three classes of parallel systems: traditional multiprocessors, chip multiprocessors, and simultaneous multithreading processors.

Use of these heuristics allows parallelization to provide acceleration for the simulators of many models. Indeed, for certain models, superlinear speedup is obtained: up to 7.56 for four processors. The clustering techniques have a larger effect than lock mitigation techniques. Recommendations for which heuristics should be used in what situation are summarized in Table 4.16. Instance-Based Clustering should be preferred for small models, while Instance-Aware Dominant Sequence Clustering performs better for medium-sized and large models. Lock rescheduling is nearly always better than lock avoidance, though the difference is usually small. Clustering (and perhaps not parallelization) should not be used for simulation of small models on simultaneous multithreading processors.

Chapter 5

Conclusions and Future Directions

In this dissertation, I have presented means to schedule model concurrency onto single and multiple processors in order to accelerate structural microarchitectural simulation. This chapter recapitulates the conclusions reached and contributions made by this dissertation and discusses possible directions for future work.

5.1 Conclusions and Contributions

This dissertation has shown that static and hybrid uniprocessor scheduling techniques can be used successfully to increase structural simulation speed for the Heterogeneous Synchronous Reactive model of computation. In particular, partitioned scheduling has been shown to be highly effective for scheduling models within the Liberty Simulation Environment. The use of static partitioned scheduling can provide speedups of up to 2.09 over dynamic scheduling.

Contrary to prior belief, partitioned scheduling can generate correct static schedules for the most common model of computation, the zero-cycle Discrete Event MoC, when models are restricted to be microarchitecturally synchronous. With modifications, partitioned scheduling can generate correct static schedules for the larger class of models which are logically synchronous if the signals can be represented in a fixed number of bits. Thus all microarchitectural models which describe synchronous designs where signals can be represented in a fixed number of bits can be statically scheduled using partitioned scheduling

Selective-trace techniques reduce the number of codeblock invocations, but generally do not provide improved performance because of the increased overhead they imply. Acyclic scheduling can be effective, providing speedups of up to 1.44. Levelized event-driven scheduling does not reliably improve performance, though it does decrease codeblock invocations.

Four enhancements to scheduling techniques are needed to make scheduling practical: dependence information enhancement to improve the precision of signal graphs used for scheduling, dynamic subschedule embedding to control scheduler execution time in the face of limited dependence information, subgraph-based invocation coalescing to reduce redundant work, and forced invocation to allow dynamic scheduling within an HSR framework.

Structural simulators using the Heterogeneous Synchronous Reactive model of computation can be automatically parallelized and that the method used for scheduling can have a large impact upon simulation speed. A simple list scheduling approach results in parallel simulators which are slower than the corresponding uniprocessor simulators.

The parallel simulator scheduling problem is an instance of the multiprocessor task scheduling problem with precedence constraints, sequence dependence, resource constraints, and communication costs. This dissertation has introduced two new lock mitigation heuristics to reduce time spent waiting for locks to meet resource constraints and two new clustering heuristics to improve cache locality and reduce the effects of sequence dependence. These heuristics have been evaluated for three classes of parallel systems: traditional multiprocessors, chip multiprocessors, and simultaneous multithreading processors.

Use of these heuristics allows parallelization to provide acceleration for the simulators of many models. Indeed, for certain models, superlinear speedup is obtained: up to 7.56 for four processors. The clustering techniques have a larger effect than lock mitigation techniques. Instance-Based Clustering should be preferred for small models, while Instance-Aware Dominant Sequence Clustering performs better for medium-sized and large models. Lock rescheduling is nearly always better than lock avoidance, though the difference is usually small. Clustering (and perhaps not parallelization) should not be used for simulation of small models on simultaneous multithreading processors.

5.2 Future Directions

This dissertation has not exhausted the possibilities for the use of scheduling to accelerate microarchitectural simulation. Opportunities exist for improvements in both uniprocessor and parallel scheduling.

For uniprocessor scheduling, dynamic subschedule scheduling could gain from better tuning of what constitute large or small strongly-connected components. Both partitioned scheduling and invocation scheduling could be adapted to include codeblock invocation time estimates, thereby avoiding repetitions of the costliest codeblocks. Automatic dependence analysis using the code of module instances would remove the responsibility for annotating modules from the user, thereby making scheduling both easier to use and more exact.

For parallel scheduling, parallelization of dynamic subschedules would be helpful when the user has not provided complete dependence enhancement. Further investigation of situations where clustering techniques assign most work to a single processor could lead to improvements in clustering or means to automatically reduce the number of processors used when model parallelism does not warrant them. Other algorithmic enhancements are also possible. One such line of inquiry involves forms of clustering which do not constrain lock mitigation as tightly, allowing both lock wait time reduction and locality improvement to be addressed together. Another involves better modeling of internal cache misses as communication costs in iDSC or a variant of MH.

Better estimates of codeblock invocation costs and communication costs would benefit all forms of scheduling greatly. These costs may be predictable via joint analysis of model structure and code size metrics. Another possibility is use of profiling information. Profiling could even be used at run-time to re-parallelize the simulator, making it adapt to host system load.

Additional studies into the scalability of the parallelization techniques to more than four processors would also be helpful in determining which heuristics should be applied when using larger systems. Another interesting direction to pursue would be the integration of thread-level automatic parallelization with job-level parallelism techiques such as DiST in systems where there is a hierarchy of parallelism, such as a cluster of machines containing chip multiprocessors.

The proof that partitioned scheduling will work for a very large class of zero-delay Discrete Event systems is quite important because it open the door for the support of multiple clock domains in the models. Multiple domains are much easier to support in the DE model of computation than the HSR MoC because there are never issues with W-codeblocks updating state, which would reduce the ability to reuse modules which did so in multiple clock models. Multiple clock domains are very important in the modeling of systems beyond just a single microprocessor and are becoming increasingly important to detailed modeling within a microprocessor.

Finally, codeblocks are not sacrosanct. Further large-scale improvements in performance may require us to "open the black box" and restructure the code of module templates. Doing so will allow improvements such as code specialization, inlining of codeblocks, downgrading of signals to variables, and other techniques which hand-written simulators take advantage of at present. Indeed, aggressive codeblock merging or partitioning could eventually lead to an automatically parallelized monolithic simulator generated from a structural model.

5.3 A Final Word

The speed of microarchitectural simulation matters. So also does the ease with which microarchitectural models can be created and modified. Structural simulation frameworks such as the Liberty Simulation Environment are a consequential step forward in improving the latter. But they will not gain widespread acceptance if the speed cannot be made competitive with that of hand-coded simulators. This dissertation represents the first few significant steps in accelerating structural microarchitectural simulation. Faster structural microarchitectural simulation promises to allow microarchitects to enjoy simulators which are both fast and easy to modify, ultimately resulting in more and better evaluation of design alternatives and better design decisions.

Bibliography

[1] Spec newsletter. http://www.spec.org.

[2] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[3] P. Agrawal. Concurrency and communication in hardware simulators. *IEEE Transactions on Computer-Aided Design*, 5(4), October 1986.

[4] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, Feb 1995.

[5] B. S. Baker and E. G. Coffman, Jr. Mutual exclusion scheduling. *Theoretical Computer Science*, 162:225–243, 1996.

[6] K. C. Barr, R. Matas-Navarro, C. Weaver, T. Juan, and J. Emer. Simulating a chip multiprocessor with a symmetric multiprocessor. In *Boston Area Architecture Workshop*, January 2005.

[7] Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, and G. M. Silberman. SLS - a fast switch-level simulator. *IEEE Transactions on Computer-Aided Design*, CAD-7(8):838–849, August 1988.

[8] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge. HSS - a high-speed simulator. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):601–617, July 1987.

[9] H. Bekić. Definable operations in general algebras, and the theory of automata and flowcharts. In C. B. Jones, editor, *Programming Languages and Their Definition*, volume 177 of *Lecture Notes in Computer Science*, pages 30–55. Springer-Verlag, 1984.

[10] H. L. Bodlaender, K. Jansen, and G. J. Woeginger. Scheduling with incompatible jobs. *Discrete Applied Mathematics*, 55:219–232, 1994.

[11] R. E. Bryant. Simulation of packet communication architecture computer systems. Master's thesis, Massachusetts Institute of Technology, 1977.

[12] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[14] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–453, September 1979.

[15] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[16] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L.-S. Peh, and M. Martonosi. Hardware-modulated parallelism in chip multiprocessors. In *Proceedings of the 2005 Workshop on Design, Architecture, and Simulation of Chip Multi-Processors*, November 2005.

[17] M. Chidister and A. George. Parallel simulation of chip-multiprocessor architectures. *ACM Transactions on Modeling and Computer Simulation*, 12(3):176–200, July 2002.

[18] N. Christofides. *Graph Theory: An Algorithmic Approach*, pages 225–235, 279–281. Academic Press Inc., 1975.

[19] P. Coe, F. Howell, R. Ibbett, and L. Williams. A hierarchical computer architecture design and simulation environment. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(4):431–446, October 1998.

[20] M. M. Denneau. The Yorktown simulation engine. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 55–59, 1982.

[21] S. A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. Ph.D. Thesis, University of California, Berkeley, 1997.

[22] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, June 1990.

[23] J. R. Ellis. *Bulldog: A compiler for VLIW Architectures*. Ph.D. Thesis, Yale University, 1985.

[24] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 0018-9162:68–76, February 2002.

[25] B. Falsafi and D. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

[26] P. Farabosci, G. Desoli, and J. Fisher. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett Packard Laboratories, December 1998.

[27] A. Ferscha. Parallel and distributed simulation of discrete event system. In A. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 1003–1041. McGraw-Hill, 1996.

[28] J. A. Fisher. *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources*. Ph.D. Thesis, New York University, 1979.

[29] R. S. French, M. S. Lam, J. R. Levitt, and K. Olukotun. A general method for compiling event-driven simulations. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, June 1995.

[30] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, October 2000.

[31] S. Gai, F. Somenzi, and M. Spalla. Fast and coherent simulation with zero delay elements. *IEEE Transactions on Computer-Aided Design*, CAD-6(1):85–92, January 1987.

[32] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, June 1975.

[33] A. D. George and J. Steven W. Cook. Distributed simulation of parallel DSP architectures on workstation clusters. *Simulation*, 67(2):94–105, 1996.

[34] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *Proceedings of the 2003 ACM SIGMETRICS Conference*, pages 1–12, 2003.

[35] D. Gracia Pérez. Personal communication to David A. Penry, January 2006.

[36] D. Gracia Pérez, G. Mouchard, and O. Temam. A new optimized implementation of the SystemC engine using acyclic scheduling. In *Proceedings of the Design Automation and Test in Europe Conference*, pages 552–557, 2004.

[37] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[38] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[39] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the European Conference on Design, Automation and Test*, March 1999.

[40] D. Hommais and F. Pétrot. Efficient combinational loops handling for cycle precise simulation of system on a chip. In *Proceedings of the 24th EUROMICRO Conference*, pages 51–54, 1998.

[41] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:22–35, 1961.

[42] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[43] S. Irani and V. Leung. Scheduling with conflicts on bipartite and interval graphs. *Journal of Scheduling*, 6:287–307, 2003.

[44] A. Jantsch. *Modeling Embedded Systems and SOC's*. Morgan Kaufmann, 2004.

[45] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[46] G. Jennings. A case against event-driven simulation for digital system design. In *Proceedings of the 24th Annual Symposium on Simulation*, pages 170–177, 1991.

[47] K. Kailas, K. Ebcioglu, and A. Agrawala. CARS: A new code generation framework for clustered ILP processors. In *Proceedings of the 7th Annual Symposium on High-Performance Computer Architecture*, pages 133–143, 2001.

[48] K. L. Kapp, T. C. Hartrum, and T. S. Wailes. An improved cost function for static partitioning of parallel circuit simulations using a conservative sychronization protocol. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 78–85, June 1995.

[49] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center*, pages 85–103. Plenum Press, 1972.

[50] H. K. Kim and J. Jean. Concurrency preserving partitioning (CPP) for parallel logic simulation. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, pages 98–105, May 1996.

[51] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw Hill, 1970.

[52] V. Krishnaswamy and P. Banerjee. Actor based parallel VHDL simulation using Time Warp. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, May 1996.

[53] V. Krishnaswamy, G. Hasteer, and P. Banerjee. Automatic parallelizaiton of compiled event driven VHDL simulation. *IEEE Transactions on Computers*, 54(4), April 2002.

[54] Y. Kwok and I. Ahmad. Dynamic critical path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[55] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), December 1999.

[56] M. S. Lam. Instruction scheduling for superscalar architectures. *Annual Reviews in Computer Science*, 4:173–201, 1990.

[57] V. S. Lapinskii, M. F. Jacome, and G. A. De Veciana. Cluster assignment for high-performance embedded VLIW processors. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):430–454, July 2002.

[58] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report TR-93-22, Sun Microsystems Laboratory, 1993.

[59] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[60] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual Symposium on Microarchitecture*, pages 111–122, November 2002.

[61] J. K. Lenstra and A. H. G. Rinnnoy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, January–February 1978.

[62] Y. Levendel, P. Menon, and S. Patel. Special purpose computer for logic simulation using distributed processing. *Bell Systems Technical Journal*, 61:2873–2910, December 1982.

[63] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. E. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1–2):51–142, 1993.

[64] Y. Lu. Scheduling of wafer test processes in semiconductor manufacturing. Master's thesis, Virginia Polytechnic Institute and State University, 2001.

[65] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design*, 13(7):950–956, July 1994.

[66] P. M. Maurer and Z. Wang. Techniques for unit-delay compiled simulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 480–485, 1990.

[67] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast and portable architecture simulator. In *Workshop on Performance Analysis and its Impact on Design (PAID)*, June 1997.

[68] M. Narasimhan and J. Ramanujam. A fast approach to computing exact solutions to the resource-constrained scheduling problem. *ACM Transactions on Design Automation of Electronic Systems*, 6(4):490–500, 2001.

[69] A. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael. Accuracy and speed-up of parallel trace-driven architectural simulation. In *International Parallel Processing Symposium*, 1997.

[70] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM Press, 1996.

[71] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[72] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

[73] S. Patil, P. Banerjee, and C. D. Polychronopoulos. Efficient circuit partitioning algorithms for parallel logic simulation. In *Proceedings of Supercomputing '89*, pages 361–370, November 1989.

[74] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, CA, 2nd edition, 1998.

[75] D. Penry and D. I. August. Optimizations for a simulator construction system supporting reusable components. In *Proceedings of the 40th Design Automation Conference (DAC)*, June 2003.

[76] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 29–40, February 2006.

[77] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[78] F. Pétrot, D. Hommais, and A. Greiner. Cycle precise core based hardware/software system simulation with predictable event propagation. In *Proceedings of the 23rd EUROMICRO Conference*, pages 182–187, 1997.

[79] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[80] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–70, February 1996.

[81] C. V. Ramamoorthy, K. M. Chandy, and J. Mario J. Gonzalez. Optimal scheduling strategies in a multi-processor system. *IEEE Transactions on Computers*, C-21:137–146, February 1972.

[82] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, volume 21, pages 17–26, 1986.

[83] S. Selvakumar and C. S. R. Murthy. Scheduling precedence constrainted task graphs with non-negligible intertask communiation onto multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):328–336, 1994.

[84] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[85] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):75–87, February 1993.

[86] G. C. Sih and E. A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems*, 4(6), June 1993.

[87] J. W. Smith, K. S. Smith, and R. J. Smith II. Faster architectural simulation through parallelism. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 189–194, 1987.

[88] S. P. Smith, B. Underwood, and M. R. Mercer. An analysis of several approaches to circuit partitioning for parallel logic simulation. In *Proceedings of the 1987 International Conference on Computer Design*, pages 664–667, 1987.

[89] G. W. Snedecor and W. G. Cochran. *Statistical Methods*. Iowa State University Press, sixth edition, 1967.

[90] L. Soulé and T. Blank. Parallel logic simulation on general purpose machines. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 166–171, 1988.

[91] L. Soulé and A. Gupta. An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4), October 1991.

[92] C. Sporrer and H. Bauer. Corolla partitioning for distributed logic simulation of VLSI-circuits. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 85–92, May 1993.

[93] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[94] S. Subramanian, D. M. Rao, and P. A. Wilsey. Study of a multilevel approach to partitioning for parallel logic simulation. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 833, 2000.

[95] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[96] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–665, June 1994.

[97] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[98] E. G. Ulrich. Exclusive simulation of activity in digital networks. *Communications of the ACM*, 12(8):102–110, February 1969.

[99] M. Vachharajani. *Microarchitectural Modeling for Design-space Exploration*. Ph.D. Thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, United States, November 2004.

[100] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.

[101] M. Vachharajani, N. Vachharajani, D. A. Penry, J. Blome, and D. I. August. The Liberty Simulation Environment, Version 1.0. *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, 31(4), March 2004.

[102] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.

[103] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The Liberty Simulation Environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems*, 24(3), August 2006.

[104] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. SSIM: A software levelized compiled-code simulator. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 2–8, 1987.

[105] X. Wang and P. M. Maurer. Scheduling high-level blocks for functional simulation. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 87–90, 1989.

[106] Z. Wang and P. M. Maurer. LECSIM: A levelized event driven compiled logic simulator. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 491–496, 1990.

[107] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report 2004-003, Computer Architecture Lab at Carnegie Mellon, November 2004.

[108] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[109] K. F. Wong, M. A. Franklin, R. D. Chamberlain, and B. L. Shing. Statistics on logic simulation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 13–19, 1986.

[110] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[111] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[112] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[113] W. H. Yu. *LU Decomposition on a Multiprocessing System with Communication Delay*. Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1984.