

© Copyright by Aqeel Asghar Mahesri, 2004

EXPLOITING PARALLELISM BETWEEN CONTROL-FLOW AND DATA  
GENERATING INSTRUCTIONS

BY

AQEEL ASGHAR MAHESRI

B.S., University of California at Berkeley, 2002

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

# ABSTRACT

In order to obtain continued performance improvements from microprocessors, ways must be found to increase the degree of parallel execution. However, the parallelism of single-threaded programs is limited by the control flow within the code. This thesis proposes a technique to reduce the control flow bottleneck by observing that much of the control flow computation can be performed in parallel with data computation. Using this observation the program can be partitioned into a control flow generating portion and a data computing portion, to be executed in parallel. This thesis defines a general execution model capable of exploiting this parallelism and describe a possible implementation of the model, consisting of a CMP with one processor core executing the control portion and one core the data portion. An evaluation shows such a technique, given substantial hardware support, offers substantial performance benefit over a conventional processor, with IPC increasing 7% to 27%, depending on the aggressiveness of the partitioning. This thesis also explore ways to extend this technique beyond two-way to wider parallelism.

To my family for their high expectations and encouragement.

# ACKNOWLEDGEMENTS

I would like to thank the members of the ACS group for their simulation support and their help with implementation details. In particular, I would like to thank Brian Fahs and Brian Slechta, whose code I use heavily, as well as Francesco Spadini for initial ideas on program partitioning.

I would also like to thank my advisor, Sanjay Patel, for his ideas as well as a dose of optimism.

# TABLE OF CONTENTS

	Page
<b>CHAPTER 1 INTRODUCTION</b> . . . . .	<b>1</b>
<b>CHAPTER 2 PROGRAM PARTITIONING</b> . . . . .	<b>4</b>
2.1 Control Thread . . . . .	4
2.1.1 Complete Control Thread . . . . .	4
2.1.2 Partial Control Thread . . . . .	5
2.1.3 Partitioning Preciseness . . . . .	7
2.2 Data Thread . . . . .	7
<b>CHAPTER 3 CONTROL THREAD SELECTION</b> . . . . .	<b>9</b>
3.1 Ideal Selection . . . . .	9
3.2 Realistic Control Thread Selection . . . . .	9
3.2.1 Compiler Selection . . . . .	10
3.2.2 Hardware Selection . . . . .	11
3.3 Modeling Control Thread Selection . . . . .	11
3.3.1 Methodology . . . . .	11
3.3.2 Benchmarks . . . . .	13
3.3.3 Selection Results . . . . .	13
<b>CHAPTER 4 PROPOSED EXECUTION MODEL</b> . . . . .	<b>18</b>
4.1 Parallel Execution Model . . . . .	18
4.1.1 Complete Control/Data Threads . . . . .	18
4.1.2 Partial Control/Data Threads . . . . .	19
4.1.3 Multiple Data Processors . . . . .	20
4.2 Example Implementation . . . . .	21
4.2.1 Instruction Fetch . . . . .	21
4.2.2 Register Communication . . . . .	22
4.2.3 Memory Speculation and Recovery . . . . .	22
<b>CHAPTER 5 PERFORMANCE EVALUATION</b> . . . . .	<b>25</b>
5.1 Simulation Environment . . . . .	25
5.1.1 Simulator Details . . . . .	25
5.1.2 Benchmarks and Partitioning Heuristics . . . . .	27
5.2 Results . . . . .	27
5.3 Discussion . . . . .	28

5.4	Limitations . . . . .	30
<b>CHAPTER 6</b>	<b>RELATED WORK . . . . .</b>	<b>32</b>
<b>CHAPTER 7</b>	<b>CONCLUSION . . . . .</b>	<b>34</b>
7.1	Future Work . . . . .	35
<b>REFERENCES</b>	<b>. . . . .</b>	<b>36</b>

# CHAPTER 1

## INTRODUCTION

Over the last several decades, improvements in manufacturing have allowed an increasing number of transistors to be placed on a single chip. For a long time, chip makers have used this increasing number of transistors to build more complex, faster processors, adding support for pipelining, superscalar, out-of-order, and speculative execution. In the near future, however, adding more of this complexity to a single microprocessor will result in a much more difficult design and verification process while yielding only a modest, and diminishing, performance gain.

As an alternative to designing ever more complex processors, chip makers have proposed placing multiple processor cores on a single chip. Such a chip multiprocessor (CMP) will allow a multiprocessor computer to be built for the same cost as a current single processor system. Moreover, a CMP will outperform a traditional multiprocessor machine on multithreaded programs by allowing the processors to communicate with much lower latency. [1]

Unfortunately, a CMP can only accelerate multithreaded code, and writing efficient and correct multithreaded programs is very difficult. As a result, most programs, including many performance-critical applications, are written with only a single execution thread. Accelerating such single-threaded programs on CMP machines requires some way of automatically extracting parallelism from the code.

Studies [2] [3] show that even single-threaded programs do have substantial parallelism available. However, [2] also shows that the amount of parallelism that can be extracted by conventional dynamic scheduling techniques is small and too fine-grained to allow scheduling of program instructions across cores in a CMP. Extracting course-grained parallelism requires reordering computation over a very large instruction window.

A major factor that limits the degree to which computation can be reordered is the program's control flow. In order to execute parallel instructions that may be far apart in program order, the machine must know far in advance the instructions that will need to be executed. Thus, computation of the control flow constitutes a bottleneck limiting the degree of available parallelism.

This thesis introduces a technique for reducing the control-flow bottleneck. This technique depends on a key observation: instructions that compute control flow can be computed in parallel with much of the data computation. Our technique uses this observation to devote one processor core to determining the control flow of the program, and one or more cores for the remaining computation.

The first step to parallelizing a program in this manner is to derive the subprograms that will run on each processor. In particular, we need a technique to separate the portion of the program that is needed to compute control flow, and to determine the set of instructions that represents the remaining work. Chapter 2 introduces this notion of partitioning of a program into the control thread and the data thread. This partitioning can be optimized in various ways depending on the available microarchitecture and compiler. The details of the partitioning are examined by Chapter 3, which focuses on the selection of the control thread and examines the balance between control and data threads.

Additionally, we need a machine to execute the partitioned program in parallel. The machine proposed in this thesis has one processing core dedicated to evaluating the control thread, and one or more cores for evaluating the remaining instructions in the data, thread. In this machine, the *control processor* directs the code executed by the *data processors* by generating Fetch Addresses for code blocks to be executed on a data processor. Chapter 4 explores the hardware mechanisms needed to perform such a parallel computation. In addition to the processor cores, any CMP executing the partitioned program needs various queuing structures for communicating control flow information from the control processor to the data processors, and for communicating values between the control and data threads. Figure 1.1 shows an example machine model of a control processor connected to one data processor.

We evaluate the performance of our technique a machine similar to the one in Figure 1.1 in Chapter 5. The evaluation explores the performance tradeoff of various methods of program partitioning. It also examines the performance impact of coupling execution among and sharing

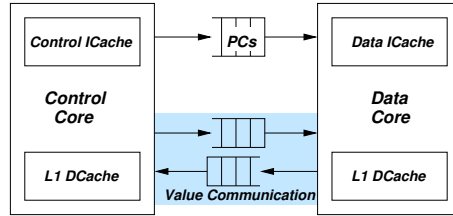


Figure 1.1: Example of a Control-Data Decoupled Machine

values between the different threads. Finally, it examines some resulting challenges in developing a wider parallel machine.

Chapter 6 examines prior work in parallelizing single-threaded applications. Finally, Chapter 7 provides concluding remarks and a discussion of future work.

## CHAPTER 2

# PROGRAM PARTITIONING

This chapter describes partitioning a program into a *control thread* and a *data thread*. The control thread is the slice of a program required to compute control flow, and the data thread is the remaining program or some superset of it.

Figure 2.1 provides a code example that is partitioned into control and data threads. In this simple loop, the two threads are highly decoupled. The control thread contains spawn instructions that initiate blocks to execute on a data processor.

### 2.1 Control Thread

This section defines the control thread. The control thread is a subset of the dynamic instruction stream that defines the control flow of the program. There are various such control threads that can be chosen from a program.

#### 2.1.1 Complete Control Thread

A *complete control thread (CCT)* consists of all of the instructions in the dynamic instruction stream that define the control flow. This includes all of the control instructions such as jumps and branches. It also includes all instructions whose outputs are needed for determining the outcome of branches and indirect jumps.

Formally, the complete control thread can be defined recursively as:

$$CCT = \{x \in DynamicInsts \mid x \text{ is a control inst} \vee \exists y \in CCT, y \text{ consumes a value produced by } x\}$$

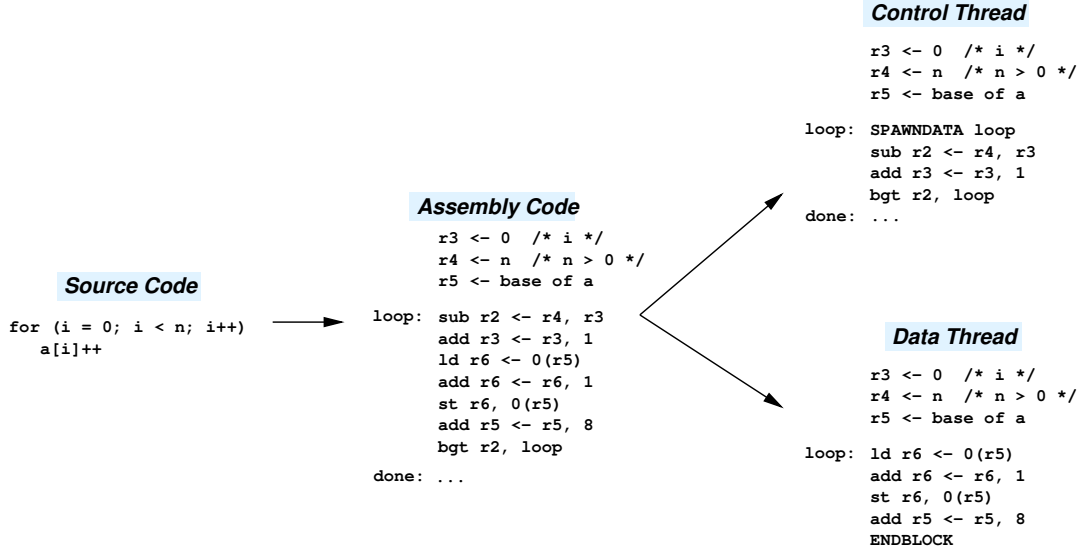


Figure 2.1: A simple example that illustrates control/data partitioning.

The control thread can also be defined as a subset of the static program. In this case, we can define the *static complete control thread (SCCT)* as the set of all static instructions that define the control flow. This includes all of the control instructions in the static program, as well as all instructions that these control instructions have dataflow dependencies on.

$$SCCT = \{x \in DynamicInsts \mid x \text{ is a control inst} \vee \exists y \in SCCT, y \text{ has a dataflow dependence on } x\}$$

Note that the projection of the *SCCT* onto the set of dynamic instructions is a superset of the *CCT*. Figure 2.2 gives an example program where the dynamic *CCT* is not equal to the *SCCT*.

### 2.1.2 Partial Control Thread

A *partial control thread (PCT)* is simply a subset of the complete control thread.

$$PCT \subset CCT$$

The partial control thread can also be defined from the static program as well. In this case, the *static partial control thread (SPCT)* is a subset of the *SCCT*.

$$SPCT \subset SCCT$$

<p><b>Source Code</b></p> <pre> ... x = foo; if (bar == 3) {   if (x == 3) {     ...   } else {     ...   } } y = (x+4)*3; ... </pre>	<p><b>Assembly Code</b></p> <pre> ... r3 &lt;- r2 /* x=foo */ r4 &lt;- 3 bne r2,r4,L2 bne r3,r4,L1 ... L1: ... L2: r5 &lt;- add r3,4 r5 &lt;- mul r3,3 ... </pre>	<p><b>Static Control Thread</b></p> <pre> ... r3 &lt;- r2 /* x=foo */ r4 &lt;- 3 bne r2,r4,L2 bne r3,r4,L1 ... L1: ... L2: ... </pre>
<p><b>Execution Trace</b></p> <pre> ... r3 &lt;- r2 /* x=foo */ r4 &lt;- 3 bne r2,r4,L2 r5 &lt;- add r3,4 r5 &lt;- mul r3,3 ... </pre>	<p><b>Dynamic Control Thread</b></p> <pre> ... r4 &lt;- 3 bne r2,r4,L2 </pre>	

Figure 2.2: A simple example that illustrates impreciseness in static control/data partitioning. Here, the instructions for computing the value of x belong in the statically precise control thread, but may not occur in the dynamic *CCT* for certain execution paths.

There are various reasons for program partitioning to select a partial rather than complete control thread. The first reason is that the precise and complete control thread might not be computable. For instance, the control thread will often have data dependencies propagated through memory. If aliasing information is incomplete, it may not be possible to determine whether some store instruction (and the instructions the store depends on) should be part of the control thread. Also, the computation of the control thread may be done on the static code rather than the dynamic code, and so the set of values that are used to determine control flow may be unknown.

A second reason to use a partial control thread would be for performance. In certain cases, some control decision may be based on a value that was computed much earlier in the program. Because the control thread and data thread processors are able to share state, the control thread will eventually see values produced by the data thread. Thus, the computation of values that will not be used by the control thread for a long time does not need to be included in the control thread. Alternatively, the data dependencies of highly biased branches can be moved to the data thread. In this case, the control thread could “guess” the control flow with high confidence, and then verify the decision when the necessary values become available from the data thread.

Finally, it may not be necessary to include all control instructions in the control thread and completely eliminate control instructions from the data thread. In this case, the control thread

would handle “global” control flow while the data thread handles “local” control flow. For instance, the data processor could handle certain simple control sequences, such as the branch in loops with no internal control flow.

### 2.1.3 Partitioning Preciseness

The partitioning of a program into control and data threads may not be exact. As noted above, any practical scheme to select the control thread is likely to miss some instructions that should be included in the *CCT*. Likewise, the scheme is also likely to include instructions in the control thread that are not in the *CCT*.

Any control thread containing instructions not in the *CCT* can be called *imprecise*. Any control thread containing instructions outside the *SCCT* can be called *statically imprecise*. Conversely, any control thread that is a subset of the *CCT* is called *precise*, and control thread that is a subset of the *SCCT* is called *statically precise*. Note that static program partitioning may produce a thread that contains the *CCT* but does not contain the *SCCT*; such a partition is still statically precise.

## 2.2 Data Thread

Like the control thread, the *data thread* can also be chosen to contain a variable number of instructions. The *minimal data thread* (*MinDT*) contains the set of dynamic instructions that are not in the selected control thread.

$$MinDT = \{x \in DynamicInsts \mid x \notin \text{control thread}\}$$

This includes instructions producing output data not used in control (hence the name data thread). The data thread also contains a substantial amount of dead code; previous work ([4] [5] [6]) has shown dead code to constitute a significant fraction, about 10%, of dynamic instructions. Since the control thread is carefully filtered, all or nearly all the dead code will be in the data thread.

The *MinDT* is defined relative to the control thread, and so it also includes any instructions

that are excluded from a partial control thread. Clearly this includes those instructions whose outputs could not be determined in advance to be necessary for determining control flow. It also includes instructions that were removed from the control thread for performance reasons. Finally, the minimal data thread must include any “local” control flow constructs excluded from the control thread.

Observe that unlike the control thread, the minimal data thread defined as a subset of the dynamic instruction stream is equivalent to the minimal data thread defined as a subset of static program, as long as the control thread is also selected as a subset of the static program.

At the other end of the spectrum, the *maximal data thread* contains the complete program. In this case, the data thread is the same as the R-stream in a Slipstream processor [7], and the control thread would be used only to generate values and control decisions for accelerating the execution of the data thread.

$$MaxDT = DynamicInsts$$

Analogous to the control thread, we can define the notion of a *complete data thread (CDT)* to be the minimal data thread plus all the instructions on which an instruction in the minimal data thread has a data dependence.

$$CDT = \{x \in DynamicInsts \mid x \in MinDT \vee \exists y \in CDT \text{ such that } x \text{ produces a value consumed by } y\}$$

Then, a *partial data thread (PDT)* is any set of dynamic instructions that is a proper subset of the complete data thread and contains the minimal data thread.

$$PDT \subset CDT \text{ such that } MinDT \subseteq PDT$$

These definitions can be extended to the case of static data thread selection as follows:

$$SCDT = \{x \in StaticInsts \mid x \in MinDT \vee \exists y \in SCDT \text{ such that } x \text{ has a dataflow dependence on } y\}$$

$$SPDT \subset SCDT \text{ such that } MinDT \subseteq SPDT$$

## CHAPTER 3

# CONTROL THREAD SELECTION

Partitioning a program involves selecting a control thread and a data thread. This chapter gives an overview of techniques for selecting a control thread. The complete control thread can be computed from a dynamic instruction trace. A real system can use either a compiler pass or some additional hardware support to compute partial control threads. A variety of schemes can be used to model the partial control threads that a realistic scheme could select.

### 3.1 Ideal Selection

The dynamic *CCT* can be computed precisely by analyzing the data dependency chain in a dynamic instruction stream. The first step is to collect a trace of the dynamic instruction stream along with all the addresses of all memory accesses. The next step is to reverse the trace. The next step is to scan the reversed trace, adding an instruction to the control thread if either it is a direct control instruction (branch, jump, etc.) or produces a value that is used in a previously encountered instruction that is also in the control thread.

### 3.2 Realistic Control Thread Selection

In a real system, the dynamic instruction trace is not available, and so it is not possible to generate the precise *CCT* or even the *SCCT*. Moreover, as pointed out in the previous chapter, there may be a performance benefit to selecting a *PCT*, as the execution time of the control thread is likely to determine the execution time of the partitioned program.

The control/data partitioning can be accomplished by dataflow analysis. This analysis can be

done either by a compiler or by the hardware at run time.

### 3.2.1 Compiler Selection

Dataflow analysis can be used to find all the dependencies for control operations. Such analysis will typically reveal all such dependencies propagated through scalar variables, and hence all dependencies through registers and many through the stack.

Finding dependencies propagated through fields in structures and through elements in arrays is more difficult. Many dependencies through structures can be identified using alias analysis. The results of alias analysis can be used to identify store instructions whose values are likely to be loaded by an instruction that is selected for the control thread. The same alias analysis results can be used to find dependencies through arrays, though this may identify too many such dependencies.

Knowing these dependencies, the compiler can select from the program the dataflow closure of the set of control instructions as the control thread. The compiler would then add explicit spawn instructions where a sequence of control thread instructions implies the execution of a sequence of data thread instructions, as well as explicit stop fetch instructions at the end of each sequence of consecutive data thread instructions.

The compiler pass can reduce the number of instructions in the control thread by filtering the set of dependencies. In particular, dataflow dependences over very long distances do not need to be included in the control thread if the parallel machine executing the partitioned program has enough time to communicate these values. Indeed, excluding such dependencies may improve performance by reducing the number of instructions in the control thread if performance is being limited by execution time of the control thread.

Compiler analyses for such a partitioning have not yet been implemented and are an area of future study. A compiler-based partitioning scheme has various potential benefits. It could lead to simpler hardware if the compiler can make certain guarantees about data sharing between control and data threads. Also, the compiler can separately allocate execution resources, in particular registers, over multiple processors, and separately schedule the different threads. However, there are also disadvantages to partitioning the program statically. In particular, the partitioning information must be visible in the program binary, which would require adding to the machine's instruction set.

### 3.2.2 Hardware Selection

The partitioning can be done dynamically in hardware using a predictor for control-flow dependence. In one possible scheme, the processor would contain a structure to record, for each instruction, whether the instruction belongs in the control thread or not. Each instruction would be marked as being in the control thread if it were a control instruction, one of its outputs were used by a control instruction, or if one of its outputs were used by an instruction previously marked as being in the control thread.

There are various advantages to partitioning a program with hardware support. Foremost among them is the greater information available. In particular, there is more accurate memory alias information available. A second advantage is that the machine's external instruction set does not need to be altered. A third advantage is that the program partitioning can be optimized for a particular machine without requiring a program to be recompiled. Likewise, hardware partitioning also has disadvantages. A major disadvantage is the complexity the implementation of partitioning would add to the hardware. Another disadvantage is the sub-optimality of running on two processors with a program optimized to use the resources of only one; for example, the program could only take advantage of having one rather than two sets of architectural registers available.

## 3.3 Modeling Control Thread Selection

As noted in the previous chapter, we have a multitude of choices in deciding what goes in the control thread. We want to evaluate the computational balance between the control and data threads, and hence the performance, that results from a variety of different control thread selections. As we have dynamic instruction traces available, we can model the control thread selection of various possible heuristics by analyzing the reversed instruction traces.

### 3.3.1 Methodology

We select the control threads through the reversed trace by considering the set of dependencies between dynamic instructions. The first control thread we want to examine is the *CCT*, the ideal case. The *CCT* is computed by scanning the reversed trace and including every direct control instruction plus every instruction whose outputs are used by an instruction that has already been

selected for the control thread.

As noted earlier, if an instruction in the *CCT* produces a value that is not used by another control thread instruction for many cycles, then it makes sense to move the instruction to the data thread and communicate the value back to the control thread processor. A partial control thread that relies on this can be computed by filtering the set of dependencies considered during the reversed trace analysis, specifically by excluding dependences with many instructions (more than the scheduling window of a typical superscalar processor) between the producer and any control thread consumers.

A compiler may have imperfect memory alias information. In particular, control thread dependencies propagated through objects stored on the heap may be difficult to find. The effect of this can be modeled by further filtering the set of dependencies to exclude memory values propagated through memory other than the stack. If the compiler has no memory alias information, or if there is good value prediction, it may see a performance advantage in completely neglecting control thread dependencies propagated through memory. This can be modeled in the reversed trace analysis by excluding all store-to-load dependencies.

As a further optimization, the compiler may want to move the dependencies of highly biased branches to the data thread, and allow the control processor to decide the direction based on speculative data. In this case, the actual data values needed for the control decision are later communicated from the data processor, and in the uncommon case where decision turns out wrong the control thread is rolled back. This optimization can be modeled by first calculating the bias of each branch in the instruction stream, and then excluding all dependencies of branches whose bias exceeds some threshold.

To further reduce the control thread, the compiler may want to move “local” control flow decisions to the data thread, while keeping only “global” control flow calculation in the control thread. One example of “local” control flow is a tight loop, consisting of a single basic block, where the loop itself does not produce any values later needed for determining control flow. This can be modeled in the reversed trace analysis by excluding any branch that goes back to the basic block containing the branch as long as no other instruction within the basic block is otherwise a control thread instruction.

These reversed trace heuristics select a subset of the dynamic instruction stream as the control

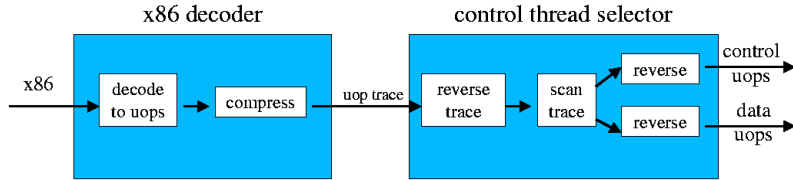


Figure 3.1: x86 decoder coupled with control thread finder.

thread. The selection heuristics can be extended to model the dynamic impreciseness introduced by static selection by maintaining a table of the PCs of all static instructions selected for inclusion in the control thread at least once in the dynamic instruction trace. All instances of instructions in the table are then included in the control thread.

The selection heuristics described above are listed in Table 3.2. The reversed trace analysis and all of the selection heuristics are implemented in the `find_control_thread` tool, which takes as input and reverses a trace of x86 instructions decoded into micro-operations. Figure 3.1 gives an overview of the decoder and `find_control_thread`. The decoder is run by `region_creator`, which takes a trace of x86 instructions, decodes it to uops (along with values), and then compresses the uops out to disk. `Find_control_thread` takes this uop and value trace, reverses it, separates the control and data threads by analyzing the data dependencies, and finally reverses the output traces. Note that the data threads produced by the tool are minimal (i.e. they have no overlap with the control threads).

### 3.3.2 Benchmarks

The control thread selection models were evaluated on 7 SPECint2000 benchmarks and 7 desktop applications from the Winstone benchmark suite. The selection of benchmarks was driven by availability. Each benchmark is a “hot-spot” trace of 50 to 100 million x86 instructions selected as being representative of the execution of the program. Table 3.1 lists the benchmarks.

### 3.3.3 Selection Results

Figure 3.2 shows the size of the complete control thread as a fraction of the total instruction stream, as well as the fraction of the control thread that remains when dependencies longer than

Table 3.1: Traces used for evaluating the control/data partitioning

benchmark	suite	size (inst)
bzip2	SPECint2000	50M
crafty	SPECint2000	50M
eon	SPECint2000	50M
gzip	SPECint2000	50M
parser	SPECint2000	50M
twolf	SPECint2000	50M
vortex	SPECint2000	50M
access	Winstone Business	100M
dreamweaver	Winstone Content	100M
excel	Winstone Business	100M
lotus notes	Winstone Business	100M
photoshop	Winstone Content	100M
powerpoint	Winstone Business	100M
soundforge	Winstone Content	100M

100 instructions are ignored. It also shows the fraction that remains when memory dependencies are propagated only through the stack. The bulk of the instruction stream (about 80%) is in the full control thread; excluding long dependencies reduces this to 68%. If memory dependencies not propagated through the stack are also excluded, the control thread falls to roughly 58% of the instruction stream; however, excluding memory dependencies over short distances would constitute a dependency violation and the speculation support in the processor would have to squash some of the control thread. Such misspeculations would occur on up to 3% of instructions if no heap dependencies are propagated. (This is the number of stores to the heap whose values are loaded by a control thread instruction less than 100 instructions in the future.) This indicates a fairly small number of resulting squashed instructions. The results also show that the control and data threads can be fairly evenly matched, which is ideal for a dual-processor machine, even with the simpler selection heuristics.

The above evaluation examines control thread selection from the dynamic instruction stream. Figure 3.3 shows the control thread fraction if the partitioning is done so as to be statically precise, using the same heuristics as in Figure 3.2. The results show that static selection adds roughly 7.5% to the control thread for each of the selection heuristics, raising the *CCT* from 80% to 86%, the max 100 from 68% to 73%, and the max 100 ignoring heap dependencies from 58% to 62%

Table 3.2: Partitioning heuristics for dynamic instruction stream and set of instructions they represent

heuristic	selected control thread
all	original program
full control thread	<i>CCT</i>
max 100	All control instructions and all dependencies with less than 100 intervening instructions
max 100 no heap	max 100 excluding dependencies propagated through the heap
max 100 no mem	max 100 excluding all dependencies propagated through memory
max 100 no tght lps	max 100 excluding loops consisting of one basic block where no value generated in the loop is used for control flow after the end of the loop
max 100 no bias	max 100 excluding all dependencies for highly biased branches (> 99% of decisions go one way)
max 100 no tght/bias	max 100 no tght lps excluding all dependencies for highly biased branches
max 100 no heap/bias	max 100 no heap excluding all dependencies for highly biased branches

of the dynamic instruction stream. Thus the dynamic impreciseness incurred by selecting the control thread statically is small, though not trivial. The control and data threads can still be reasonably matched in size. Moreover, the size of the dynamic impreciseness is consistent, with the size increase staying below 15% for all selection heuristics on all benchmarks, so static selection will not dramatically hurt performance versus an equivalent dynamically precise selection.

Figure 3.4 shows the fraction of the control thread that remains when it is selected using one of the aggressive heuristics. If all dependencies propagated through memory are ignored, the control thread falls to 53% of the dynamic instructions. As noted earlier, this means some memory accesses will cause dependence violations between the control and data threads; in this case, such violations will occur on up to 8% of instructions, possibly a large fraction of loads.

Figure 3.4 also shows the fraction of the control thread that remains if the data processor is used to execute some control instructions. In particular, if the branches on tight loops that are otherwise completely in the data thread are also moved to the data thread, the control thread falls to 58% of the execution stream, while if the instructions determining highly biased branches are moved, the control thread falls substantially to 34% of the instruction stream. If both types of

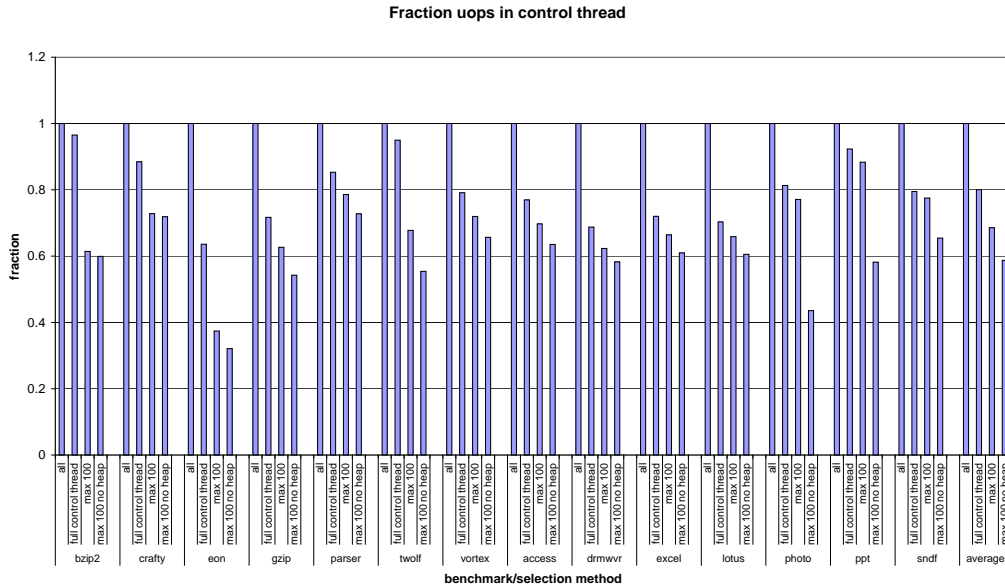


Figure 3.2: Fraction of uops in control thread with various basic selection heuristics.

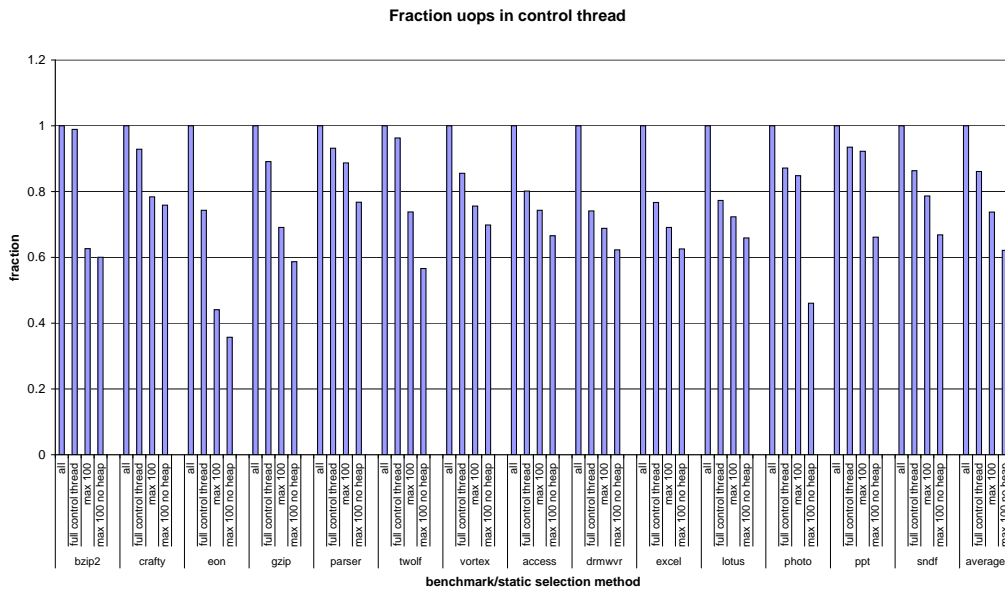


Figure 3.3: Fraction of uops in control thread with various selection heuristics, with the selection done on static instructions.

Table 3.3: Partitioning heuristics for static program and set of instructions they represent

heuristic	selected control thread
all	original program
full control thread	<i>SCCT</i>
max 100	All control instructions and all dependencies with less than 100 intervening instructions in any dynamic execution path
max 100 no heap	max 100 excluding dependencies propagated through the heap

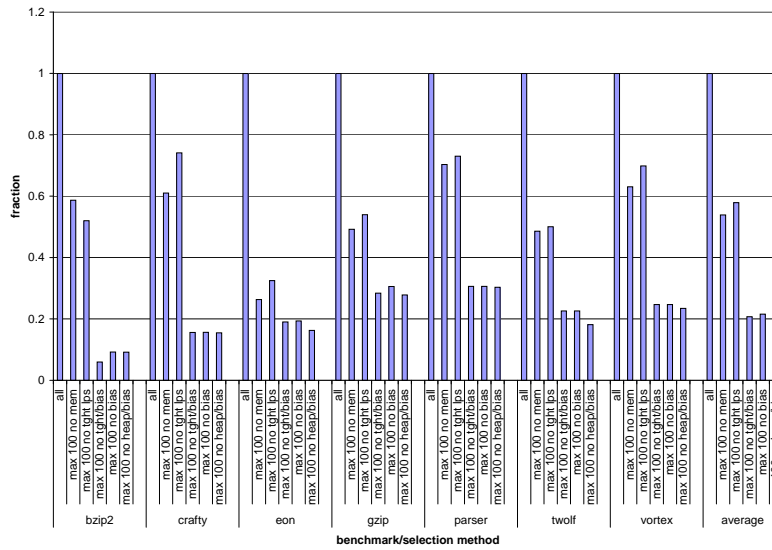


Figure 3.4: Fraction of uops in control thread with various aggressive selection heuristics.

simple control are moved, the control thread is just 23% of the instruction stream. The results here show that having a balanced load between control and data processors when there are multiple data processors will require the data thread to compute some of the simple control flow.

# CHAPTER 4

## PROPOSED EXECUTION MODEL

This chapter explores the requirements for correct, parallel execution of control and data threads. It first examines how the thread selection method affects the need for interprocessor communication and hence the hardware requirements. It also describes a dual-core CMP that meets these requirements. The CMP contains additional state needed to speculate on data dependences between the threads and recover on a data dependence violation.

### 4.1 Parallel Execution Model

The parallel execution of control and data threads requires two or more processors, with one for the control thread and at least one for the data thread. The control thread must be able to communicate control decisions to the other processor(s). Moreover, if either the control or data thread are partial, there must be additional mechanisms for communicating data between the processors. The execution model is shown conceptually in Figure 4.1.

#### 4.1.1 Complete Control/Data Threads

In the control/data parallel execution model, the control processor decides the control flow of both the control and data threads. As such, the control processor needs to send to the data processor, after each jump or branch instruction, the next set of addresses or next set of instructions that the data processor needs to execute. The data processor fetches the instructions specified by the control processor rather than by an internal PC. This is shown conceptually in Figure 4.1a.

In the ideal case where we have a complete control and data thread (not a maximal data thread),

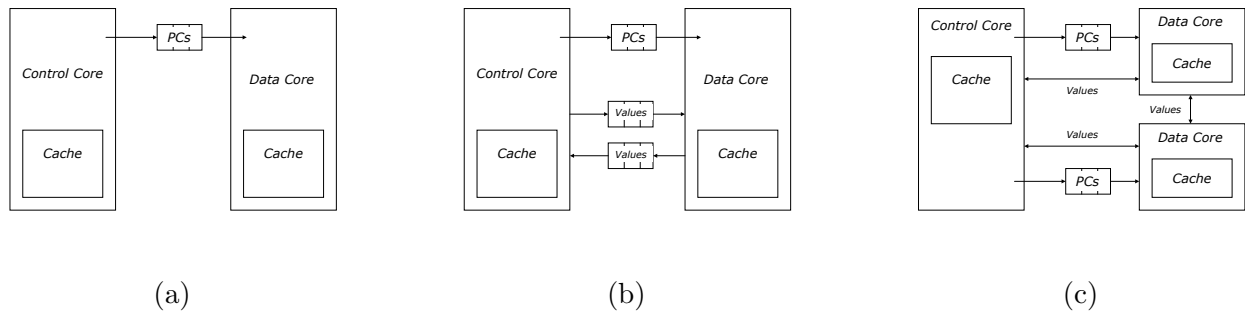


Figure 4.1: Conceptual execution models: a. complete control and data threads; b. partial control and data threads; c. partial control and data threads with data thread distributed over multiple processors.

there is no need to communicate data values between the two threads. All the values needed by the control thread are produced by instructions within the control thread, and likewise for the data thread.

Despite the simplicity of the required execution model, a real system would not execute complete control/data threads. First, the complete control thread is only 20% smaller than the original instruction stream (14% smaller for static selection), and it has a higher density of branches and dataflow dependencies, so the IPC will be lower, putting a fairly tight upper bound on performance. Second, the data thread has many replicated instructions that duplicate computation done by the control thread; sometimes, these values are more easily communicated than recomputed. Finally, actually computing complete control and data threads is impossible without knowing in advance the dynamic execution sequence. As a result, any real system would have to support partial control/data threads.

#### 4.1.2 Partial Control/Data Threads

The hardware requirements for correct execution increase substantially when one or both threads is partial. In particular, the values computed by one thread may need to be communicated to the other. Moreover, the communication must be carried out at a specific point in the instruction streams, to preserve data dependences. The machine requirements are shown in Figure 4.1b.

When the data thread is partial, values will need to be communicated from the control to

the data processor. Because instructions executing in the data thread are initiated by the control thread, we can guarantee that the data thread is always executing with or behind the control thread. As a result, we can communicate values from the control to the data processor fairly simply, and the threads can be selected such that dependence violations on values produced by control and consumed by data do not occur.

When the control thread is partial, control decisions may depend on values produced by the data thread. Thus, in addition to the mechanisms needed to execute the control and data threads when the control thread is complete, we now need mechanisms to allow communication of values from the data to the control thread. At each instruction, the control processor will need to get the most recent value for any input registers or memory locations, and so will need to check whether the values in control processor's internal state have been updated more recently by the data thread. This requires both a communication network and some mechanism for determining which value is more recent in the program order.

Moreover, because the control and data threads may not be executing synchronously, the data thread may not have produced certain values that are needed for control decisions at the time those control decisions are made, and such control decisions will need to be made speculatively. The two basic requirements for enabling speculative control threads are the same as for other speculative multi-threading: a mechanism to detect sharing violations and a mechanism to undo speculative state updates. However, the specific requirements for an undo mechanism are more complex, for two reasons. First, the misspeculation may require undoing part of the execution of both the control and data threads, as the data thread may have executed incorrect instructions based on incorrect control decisions. Second, neither control nor data thread can be completely squashed; instead, both threads have to be rolled back to the same point in the program.

### **4.1.3 Multiple Data Processors**

Conceptually, it is possible to further distribute the data thread itself over multiple processors. In this situation, the control thread would determine the distribution of data thread instructions to different data processors. In this case, we still need the same structures as for partial control/data threads, except these need to be extended to multiple processors. This is shown in Figure 4.1c.

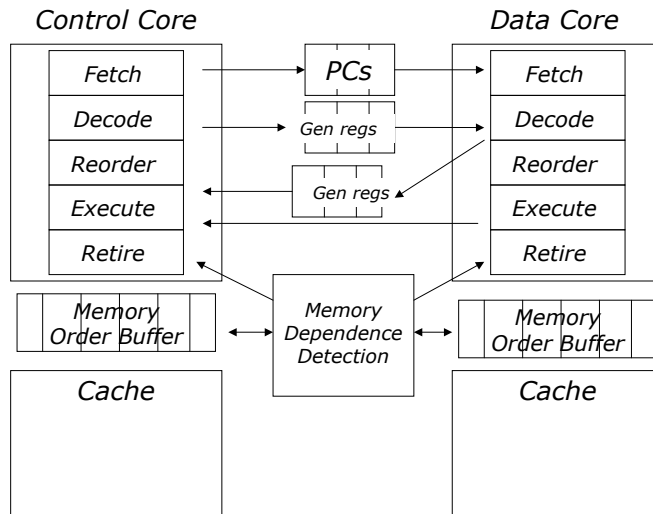


Figure 4.2: Diagram of proposed CMP.

## 4.2 Example Implementation

The conceptual execution model in Figure 4.1b, for partial control and data threads, can be implemented in the dual-core CMP described below. The machine consists of two standard, out-of-order processing cores connected by the communication network described in detail below.

Figure 4.2 gives an overview of the proposed CMP. The CMP includes the two cores plus a FIFO containing fetch addresses for the data core, some structures for register communication, and a structure for maintaining correct ordering of memory operations.

### 4.2.1 Instruction Fetch

In the execution model of Figure 4.1b, the processor core executing the control thread drives the fetch order of the data thread core. The implementation of this is fairly straightforward.

When the control core decodes a *SPAWN* instruction, it sends the target of the spawn instruction to the data core. The data core buffers all such spawn requests in a FIFO. At the fetch stage, the data core initiates fetch from the location of the spawn. If the data core encounters a *ENDBLOCK*, it initiates fetch from the next spawn in the FIFO.

One complication arises from branch mispredictions. Since the spawn requests are initiated in

the fetch/decode stages of the pipeline, the requests are speculative. When the branch direction is determined, the control core must squash its own incorrect speculative instructions and must also signal the data core to squash its corresponding, incorrect speculative instructions.

### 4.2.2 Register Communication

All register communication between the data and control threads can be identified explicitly. Mechanisms for handling register communication were developed in [8] and [9].

Our register communication mechanism differs in the asymmetry between the control and data processors. Both processors maintain state (a register update bitmask) recording the set of architectural registers that are being updated by the current group of instructions. The control processor, when it encounters a spawn instruction, sends this information to the data processor along with the next fetch PC, and resets its register update mask. From this, the data processor knows which registers in its own register file are current and which need to be updated from the control processor, and any instructions needing registers generated by the control thread remain in their reservation stations until the value becomes available.

The data processor, when it encounters an end block instruction, sends its register update mask to the control processor, along with an identifier for the originating spawn instruction. The control processor updates its register file by reading all the new values produced by the data thread. It also checks to see whether instructions in the control thread incorrectly read any stale values in the updated registers; if so, it replays any such instructions.

This register communication mechanism requires the compiler (or hardware partitioner) to guarantee that register values generated by the data thread do not get used by the control thread for at least an instruction window's number of instructions. Otherwise, the control processor will repeatedly need to replay instructions, hurting performance. This is a reasonable guarantee for the compiler to make, since all register communication can be identified explicitly in the static program.

### 4.2.3 Memory Speculation and Recovery

When the control thread attempts to read data generated by the data thread, it may or may not get the correct data. Whether the control thread gets the correct data depends largely on the timing

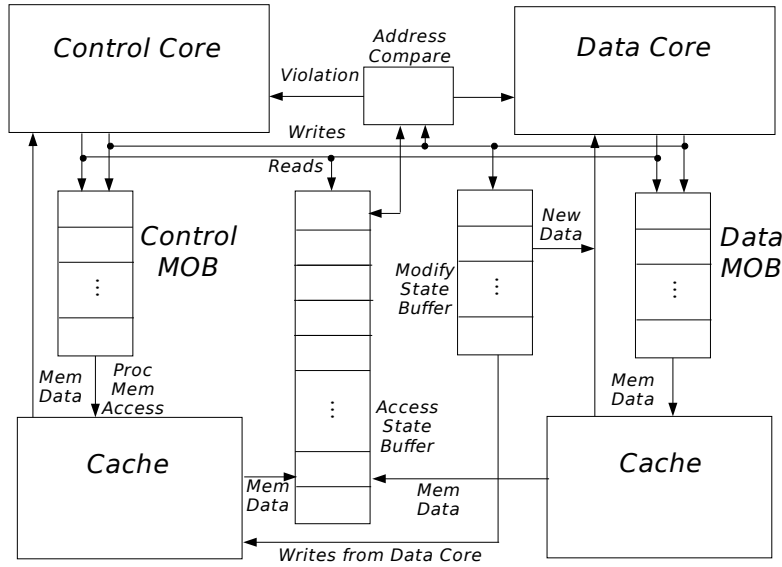


Figure 4.3: Detecting memory dependence violations.

of the two threads. Correct, high-performance execution requires timely detection of incorrect data loads as well as quick recovery.

### Detecting Violations

A mechanism for detecting data dependence violations is proposed by [9]. This mechanism consists of a memory disambiguation table (MDT), which works in a manner similar to a directory in a shared-memory multiprocessor. The MDT is sufficient for detecting violations. However, it is insufficient for the design proposed here, as we need to know exactly *where* a violation occurs in order to roll back the threads to the correct point.

To detect data dependence violations, the control/data processor uses the structures shown in Figure 4.3. The structures consist of two tables, the modify state buffer and the access state buffer, and some logic that examines these tables for dependence violations. Both the MSB and ASB store the memory address, logical instruction sequence number, and data value for each store and load, respectively. Both the MSB and the ASB are CAM structures, with the memory address used as the index.

When a load is performed by one of the threads, the address and value of the load are put in the

ASB, along with a sequence number for the immediately preceding spawn instruction. The load also searches the MSB for all stores to the same address; if one is found with a sequence number earlier than the current load, its value is forwarded to the processor.

When a store is performed, the address and value of the store are forwarded to the cache for the other processor and also put in the MSB, along with the sequence number. A lookup is performed in the MSB for all other stores with the same address; the one with the smallest sequence number greater than that of the current store is selected. Also, a lookup is done in the ASB for all loads with the same address. This set is then searched for all loads with sequence numbers between the current store and the selected store. For all such loads that are found, the sequence number and value are compared. If the sequence number of the load is greater than that of the store, and the data value is different, there has been a dependence violation.

## Handling Violations

If a memory dependence violation is detected, we must be able to roll back the state of both processors to the point of the violation. This can be done using the standard hardware, such as the ROB, that each processor uses for speculation.

In order to be able to roll back to the point of any memory dependence violation, we must keep a memory operation in the reorder buffer until the detection hardware has guaranteed that the operation will not cause a violation. This can be accomplished by committing memory operations in the *original program order*. Thus, a load or store instruction in the control thread can commit only when all logically earlier load or store operations in the data thread have also been committed.

The data thread may be running many instructions behind the control thread, and the commit mechanism requires the data thread to execute memory operations before the control thread commits subsequent operations. As a result, the commit stage of the control processor may lag many instructions behind the execution. The performance effects of this can be alleviated by using a large reorder buffer.

The commit order requirement also effectively couples the control and data processor to be running instructions at most a reorder buffer size apart. This facilitates the exclusion from the control thread of long dependence distances discussed in Chapter 3

# CHAPTER 5

## PERFORMANCE EVALUATION

The performance of the processor proposed in Chapter 4 was evaluated on the benchmarks and partitioning heuristics from Chapter 3. The evaluation was performed on an x86-based timing simulator that implements spawn/endblock synchronization, memory communication, and stalls due to data sharing violations, but not register communication or control misspeculation. The purpose of the evaluation is thus to obtain the approximate speedup that could be obtained by the proposed processor as well as the relative speedup obtained by different partitioning heuristics.

### 5.1 Simulation Environment

The environment used to evaluate the performance of control/data partitioning is a modified version of `sim-beta`, a uniprocessor microarchitectural simulator developed for the `rePLay` project [10]. `Sim-beta` implements the x86 ISA by executing a set of RISC-like micro-operations derived from decoded x86 instructions. `Sim-beta` is trace-driven, and it can execute from the x86 trace directly or it can execute a pre-decoded uop trace generated by `region_creator` or `find_control_thread`. To simulate a CMP, two uniprocessor simulators are instantiated, with one running the control thread trace and the other the data thread trace.

#### 5.1.1 Simulator Details

The control thread simulation maintains the access state buffer to detect instances when the threads update memory in an incorrect order. This buffer stores the cycle count, instruction sequence number (from the original program), and memory address of each load by the control thread.

The control thread simulation also maintains the modify state buffer, containing the cycle count, instruction sequence number, and memory address of each store performed by the control or data thread. Using this information, the simulator can detect data dependence violations, where a load within the control thread returns an incorrect value because the value is updated by an earlier (in original program order, not time) instruction in the data thread. The simulator cannot detect when a load by the data thread returns an incorrect value. This case occurs rarely and thus minimally affects timing because the partitioning heuristics guarantee that the data thread lags behind the control thread, at least at the instruction fetch stage. However, a real machine would need to be able to detect these violations as well.

When such a dependence violation is detected, the machine is stalled by some number of cycles, determined by the number of cycles the incorrect instructions required for execution. This attempts to approximate the number of execution cycles lost due to incorrect execution resulting from the dependence violation; the simulation does not actually model the incorrect execution. If no data dependence violations are detected within the maximum allowed dependence distance from a memory access, then it is guaranteed that no earlier (in program order) memory access will occur later (in time) than the current one. In this case, the corresponding entry in the memory dependence buffer is removed and the memory access committed.

As noted in the previous chapter, a mispredicted branch may cause the control processor to issue an incorrect spawn along the incorrect execution path. The simulator does not model wrong-path execution, and so it does not model any incorrect spawns. Rather, the simulator approximates the timing effects of wrong-path execution by stalling both the control and data processors until a mispredicted branch is resolved.

The simulator also does not model register communication between the cores. All register values are available from the instruction traces being executed. However, this has minimal impact on the simulation timing because the selection of the control thread, for the selection heuristics being evaluated in this chapter, guarantees that register values generated by the data thread are available many instructions before they are needed by the control thread. Moreover the execution model already ensures that register values generated by the control thread and used by the data thread are available before they are needed by the data thread.

Table 5.1 gives the machine parameters used in the simulation.

Table 5.1: Simulated processor parameters (per processor core)

processor parameter	value
Issue/retire width	8
Number of in-flight instructions	512
Executions units	6 simple integer, 2 complex integer, 4 load/store, 3 FP
Instruction fetch	16384 uop trace cache with 8 uops/line
L1 Cache	8KB instruction and 64KB data, 1 cycle latency
L2 Cache	1MB, 10 cycle latency
Branch prediction	1K entry BTB plus gshare with 18 bit history

### 5.1.2 Benchmarks and Partitioning Heuristics

The performance was evaluated for the first 20 million macro instructions of each benchmark from Table 3.1. The simulator has memory leaks which limit the execution to between 25 and 40 million uops, so a simulation of 20 million macro instructions is the longest that will consistently complete and print out execution statistics.

For each benchmark, several of the control thread selection models from Chapter 3 were used, including the full control thread, max 100, and max 100 no heap. The control threads selected by these heuristics were coupled with their corresponding minimal data threads. For the full control thread, the control thread runs ahead of the data thread, with no limit on the distance between the two. For max 100, the control thread may only run 100 instructions ahead of the data thread, at the fetch stage. However, if this restriction is followed, there are almost no data dependence violations between the threads. In the case of max 100 no heap, data dependences propagated through memory not on the stack are excluded from the computation of the control thread. However, these will cause dependence violations, and would require some re-execution.

## 5.2 Results

Figure 5.1 shows the instructions per cycle for the original processor and the control/data dual processor, with each of the selection models. Figure 5.2 shows the IPC relative to that of the base processor.

The IPC average over the benchmarks of the base processor is 1.74 uops/cycle (each original

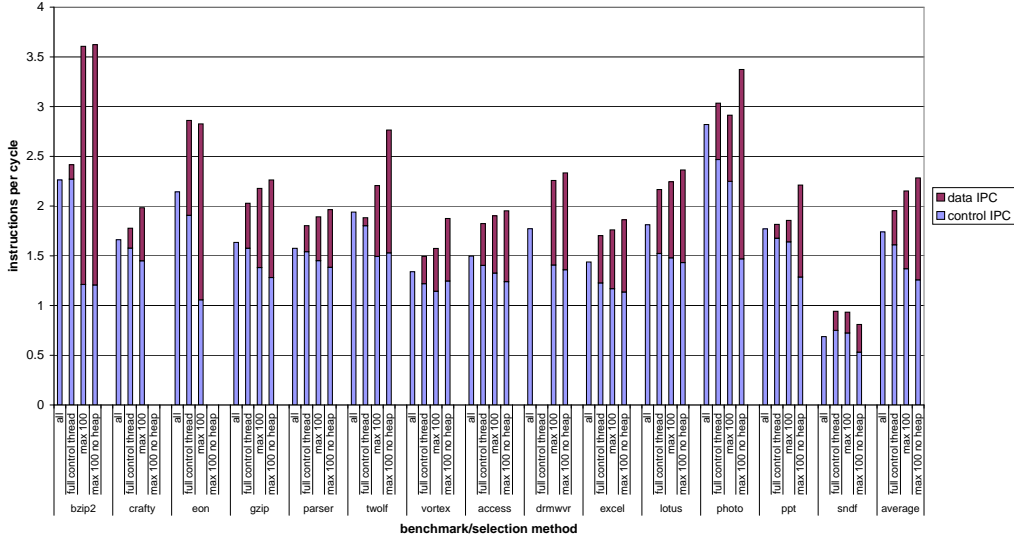


Figure 5.1: IPC of base system and control/data parallel system.

instruction decodes to roughly 1.3 uops). The IPC of the complete control thread is 1.61, while that of the corresponding minimal data thread is 0.34. Removing dependencies from the control thread reduces the parallelism available in the control thread, leading to an IPC of 1.37 and 1.26 when the partial control thread excludes long dependencies and values propagated through the heap, respectively. The overall IPC, however, rises from 1.95 for the full control thread to 2.28 when excluding long dependencies to 2.15 when excluding heap dependencies.

The corresponding increases in IPC, with the average calculated as a harmonic mean, range from 7% for the full control thread to 27% when heap dependencies are excluded.

### 5.3 Discussion

Overall, the performance improvement over differing control thread selections reflects the shortening of the control thread. The results show that, at least for most benchmarks, the instructions in the control thread are indeed the ones limiting performance.

The performance increase between the full control thread and the max 100 case is not surprising. With the control thread defining the execution critical path, removing instructions whose outputs are not used for many cycles should only help performance.

The performance increase from removing heap dependencies is somewhat surprising. Some of these dependencies will cause memory dependence violations and may stall the processor for

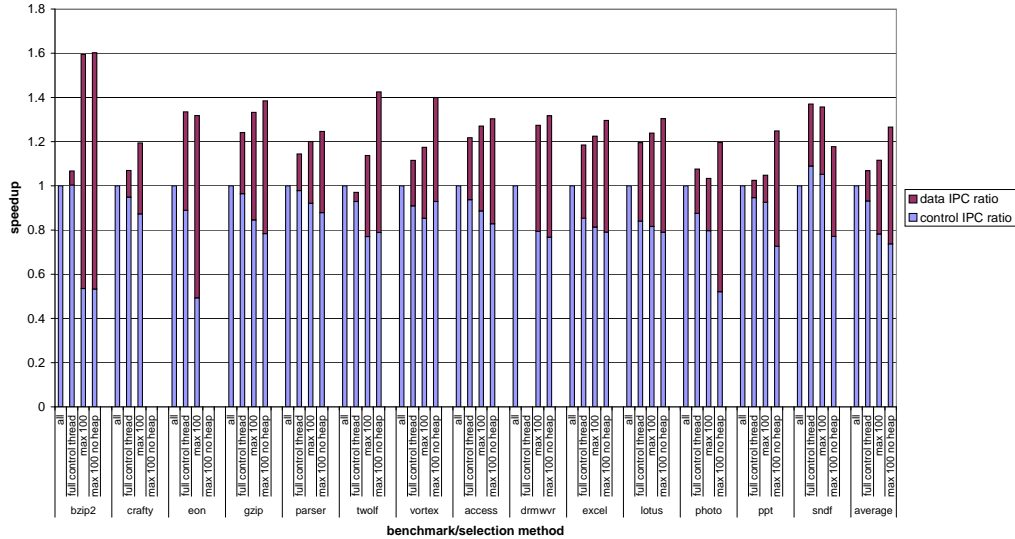


Figure 5.2: Ratio of IPC of control/data parallel system over base processor.

some time, decreasing performance. Indeed, some benchmarks show exactly this behavior, notably soundforge. However, most benchmarks show an overall improvement in performance despite having some misspeculations on memory data. One reason that memory dependence violations are still fairly rare. As noted in Chapter 3, such violations can occur on at most 3% of instructions, the loads from non-stack memory locations that are present in the max 100 control thread but not in the max 100 no heap control thread, whereas this reduces the control thread is 10% of instruction. However, for some benchmarks the performance gain is even larger than the difference between the reduction in the control thread and the number of possible violations, indicating that many of the 3% of instructions that are “risky” loads do not cause a violation. This means that the threshold of 100 intervening instructions for excluding control thread dependencies is too high, at least for this specific microarchitecture, and there is some gain from moving more instructions out of the control thread even at the expense of occasional dependence violations.

One benchmark merits further remarks. Eon shows a decrease in performance going from the full control thread to the max 100 case. This is because, as shown in Figure 3.2, eon has an exceptionally small control thread. As a result, this benchmark is bound by the execution of the data thread; the control thread is frequently stalled waiting for the data thread to catch up.

## 5.4 Limitations

The validity of any experiment based on simulation is limited by the level of detail modeled by the simulation infrastructure. Limitations in the case of sim-beta occur mainly when the model of a specific function is unimplemented or approximated. The implementation of a multiprocessor simulation atop sim-beta includes some unimplemented and approximated functionality. One approximated function is the modeling of speculation on data dependences. As noted above, the simulator does not model in detail the rollback and re-execution of threads due to dependence violations. Rather, the simulator will stall for some number of cycles to model the cost of re-executing code, with the length of the stall equal to the number of cycles required to execute the code the first time. This model of re-execution is likely to be pessimistic, as it does not account for the prefetching effect. In speculative multithreading, an incorrect speculative thread can warm up the cache and branch predictor, causing a later, correct execution of the code to be much faster. Another area of approximation is interprocessor communication through memory. All such communication is assumed to have a 5 cycle latency, while in practice the latency is dependent on various factors such as the size of the ASB/MSB and the pipeline depth. Moreover, the effect of dependence violations between a control thread store and data thread load are not modeled; although these violations are rare, as pointed out above, they would still have a small effect on timing. The major area of unimplemented functionality is register communication between the control and data cores, though it was explained earlier that the selection heuristics evaluated in this chapter prevent this from affecting timing for the control thread, and hence overall performance.

The base simulator, the uniprocessor sim-beta, also included unimplemented or approximated functionality. Examples of the former include complex and SIMD instructions in the x86 architecture. When such unsupported instructions are encountered, sim-beta will clear the machine pipeline, copy the state changes caused by the instruction into the machine state, and then restart the pipeline. Unsupported instructions, in practice, constitute roughly one of every 1000 instructions. Examples of approximated functions include those due to the trace-driven simulation mechanism. For instance, the effects of a branch misprediction are not modeled in detail; rather, the simulator assigns a misprediction penalty based on the state of the pipeline.

The simulator itself was a resource-intensive multithreaded program. As a result, simulations

were limited to 20 million macro instructions in order to get consistent results across all the benchmarks with the limited computing resources available. The short length of each simulation cannot be taken as representative of the behavior of the complete benchmark. However, because each simulation was run on a program “hot spot,” the overall results should be on the same order as the overall results from running complete benchmarks.

## CHAPTER 6

# RELATED WORK

This work extends a large body of work seeking to employ multiple processors to accelerate single-thread execution. Much of the work centers around speculative multithreading, where the machine or compiler forks threads speculatively to execute portions of a program, squashing threads whenever there is a dependency violation. In such speculatively multithreaded machines, there is one non-speculative thread representing the committed state of the machine and one or more speculative threads. An alternative approach, more similar the model proposed in this thesis, is to have one speculative thread and one or more verification threads.

One of the first proposals for decoupled architectures was the Decoupled Access/Execute Architecture [11]. In the DAE architecture, the memory address stream was decoupled from the execution stream, thereby enabling memory referencing to slip ahead of execution, providing a prefetching effect. Our architecture applies the same ideology to the branch control stream.

Another early attempt at speculative parallelization is the Multiscalar processor [12] [8]. In a Multiscalar processor, a combination of the compiler and the hardware divide a program into a number of tasks, each of which is distributed to a processor on a CMP. Each task then executes on its own processor, while the hardware ensures that all tasks see the same logical register file and the same memory. Moreover, each process executes its load/store operations speculatively, with the hardware responsible for undoing any execution resulting from an incorrect load. Multiscalar processors show large increases in IPC, though the need to synchronize register contents requires substantial additional hardware.

More recent proposals for speculative multithreading, including [13], [9] and [14], propose mechanisms that require less hardware overhead. In [13], Olukotun et. al. describe how to support fine-

grain speculative parallelism on the Hydra CMP. In [9], Krishnan and Torrellas propose a CMP design that adds a modest set of structures for speculative multithreading, specifically a synchronization scoreboard for register communication and a memory disambiguation table for detecting dependency violations among memory accesses. This design differs from Multiscalar in loosening the appearance of a single register file, using a software tool to find where registers are shared. In [14], Steffan et. al. describe an approach that imposes even less hardware overhead, by extending cache coherence mechanisms to detect dependence violations.

Whereas the preceding approaches have a non-speculative main thread and one or more speculative threads running ahead, the master-slave speculative parallelization approach [15] has one main thread that operates on speculative data and one or more less-speculative threads that run in parallel and verify the data. In the implementation of MSSP that [15] evaluates, the main thread is a distilled version of the program with many infrequent code paths removed, along with any corresponding control instructions. This main thread thus focuses on data computation while the slave threads are responsible for verifying much of the control flow. The MSSP approach is basically the reverse of the approach presented in this thesis, where the main thread computes the control flow and the slave thread does any remaining data computation.

The Slipstream processor [7] is another approach to accelerating sequential code on a CMP. The Slipstream processor takes a single-threaded program and creates two threads, called the *advanced stream* (A-stream) and *redundant stream* (R-stream). The A-stream is the original program filtered to remove as many unnecessary instructions as possible, while the R-stream consists of the complete original program. The A-stream runs faster than the original program because many instructions are removed from it, while the R-stream is accelerated by getting intermediate computation results in advance from the A-stream. The approach in this thesis takes the Slipstream approach much further, by doing much more filtering of the thread running in advance and also removing redundant instructions from the thread running behind.

## CHAPTER 7

# CONCLUSION

Control flow is a major bottleneck limiting performance. This thesis presents a method for partitioning a program in order to compress the computation of control flow and thus achieve increased parallelism.

The program is partitioned into a portion that computes the control flow and a portion that computes output data. The portion of a program that computes the control flow, the complete control thread, is typically about 80% of the total dynamic instruction stream. However, much of this can be filtered out by restricting the set of dependencies that are considered when generating the control thread. If hardware support for doing so is available, the control thread can be reduced even further by moving certain “local” control instructions to the data thread.

The partitioned program can be run on a modified dual-core CMP, with the control thread running on one core and the data thread on the other. Simulation results show that such a machine can achieve a substantial speedup over a single-processor system. The results here show average IPC increases of 7% to 27% depending on the method of partitioning the program.

The results show that, for the partitioning methods simulated here, the control-flow computation remains the critical path through the program. Higher performance can thus be achieved by moving more computation to the data thread. We show that, if simple control-flow constructs can be excluded from the control thread, it can be shrunk to as little as 20% of the instruction stream.

## 7.1 Future Work

Future work deriving from the ideas presented in this thesis will include further enhancement of both the program partitioning algorithms and the execution model, as well as a practical implementation of both.

The future work includes implementing program partitioning in a compiler. The partitioning can be done using dataflow analysis. One challenge in a compiler algorithm for control/data partitioning is filtering the set of program dependencies in order to reduce the size of the control thread without requiring excessive communication between the processors. Another challenge in the compiler is scheduling the control and data threads across two processors. The system evaluated here simply inserts spawns whenever a control thread instruction is followed by a data thread instruction but otherwise maintains the instruction ordering. However, this undermines the instruction schedules generated for uniprocessor execution.

The performance obtained by control/data partitioning is not only dependent on the partitioning algorithm but also on the microarchitecture used to execute it. As such, future work will include evaluating different processor models than the one presented here. The possible microarchitecture changes include tuning the execution resources available to the two processor cores to optimize one for control thread and the other for data thread execution.

The long term goal is to be able to fully exploit the most aggressive models for reducing the control thread, the ones that include moving “local” control flow to the data threads. In this case, the data thread is likely to be the performance bottleneck. However, previous work has shown that eliminating control flow can greatly increase the available parallelism, and so the data thread is itself likely to be highly parallelizable. Distributing the data thread over multiple processors is thus another area of future work.

# REFERENCES

- [1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [2] D. W. Wall, “Limits of instruction-level parallelism,” in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 176–188.
- [3] T. M. Austin and G. S. Sohi, “Dynamic dependency analysis of ordinary programs,” in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 342–351.
- [4] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, et al., “A performance characterization of a hardware mechanism for dynamic optimization,” in *Proceedings of the 34th Annual International Symposium on Microarchitectre*, 2001.
- [5] E. Rotenberg, “Exploiting Large Ineffectual Instruction Sequences,” North Carolina State University, Tech. rep., November 1999.
- [6] S. S. Lumetta and S. J. Patel, “Characterization of essential dynamic instructions,” in *Proceedings of SIGMETRICS 2003*, 2003, pp. 308–9.
- [7] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, “Slipstream processors: Improving both performance and fault tolerance,” in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
- [8] G. Sohi, S. Breach, and T. Vijaykumar, “Multiscalar processors,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

- [9] V. Krishnan and J. Torrellas, “A chip-multiprocessor architecture with speculative multi-threading,” *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 866–880, September 1999.
- [10] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. A. Muthler, J. Quek, et al., “Dynamic optimization of micro-operations,” in *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, 2003.
- [11] J. E. Smith, “Decoupled access/execute computer architectures,” *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 289–308, November 1984.
- [12] M. Franklin, “The multiscalar architecture,” Computer Sciences Department, University of Wisconsin - Madison, Tech. Rep. 1196, November 1993.
- [13] K. Olukotun, L. Hammond, and M. Willey, “Improving the performance of speculatively parallel applications on the hydra CMP,” in *International Conference on Supercomputing*, 1999, pp. 21–30.
- [14] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, “A scalable approach to thread-level speculation,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.
- [15] C. Zilles and G. Sohi, “Master/Slave Speculative Parallelization,” in *Proceedings of the 35th Annual International Symposium on Microarchitecture*, 2002.