September 2005

UILU-ENG-05-2214
CRHC-05-05

EXPLOITING PARALLELISM BETWEEN CONTROL AND DATA COMPUTATION

Aqeel Mahesri and Sanjay Patel

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimates or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 2005

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
Exploiting Parallelism Between Control and Data Computation

5. FUNDING NUMBERS

6. AUTHOR(S)
Aqeel Mahesri and Sanjay Patel

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Coordinated Science Laboratory
University of Illinois
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-05-2214
(CRHC-05-05)

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

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 paper 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, called the control thread, and a data computing portion, broken up into work threads. This paper further proposes an architecture to execute the control and work threads in parallel. After a detailed microarchitecture simulation, we observe a perfomance speedup of 1.03 to 1.48 on integer benchmarks, with an average of 1.16 on a dual 4-wide versus a conventional 4-wide superscalar processor, and an average of 1.20 on a dual 8-wide versus a single 8-wide superscalar processor.

14. SUBJECT TERMS multithreaded architecture, parallelization, decoupled architecture, program analysis

15. NUMBER IF PAGES
22

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Exploiting Parallelism Between Control and Data Computation

Aqeel Mahesri Sanjay Patel
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{mahesri,sjp}@crhc.uiuc.edu

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 paper 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, called the control thread, and a data computing portion, broken up into work threads. This paper further proposes an architecture to execute the control and work threads in parallel. After a detailed microarchitecture simulation, we observe a perfomance speedup of 1.03 to 1.48 on integer benchmarks, with an average of 1.16 on a dual 4-wide versus a conventional 4-wide superscalar processor, and an average of 1.20 on a dual 8-wide versus a single 8-wide superscalar processor.

1. INTRODUCTION

A major impediment to improving single thread performance is program control flow. Improving performance via parallelism within a single thread requires concurrent execution of instructions that may be far apart in program order. Control flow limits the ability of automatic (both static and dynamic) techniques for finding such independent instructions. This paper evaluates a technique for reducing the control-flow bottleneck via *decoupling* of the decision-making logic of a program from the actual work carried out by the program. The control flow portion is separated from the non-control flow portion into a *control thread* and two separate, mostly independent threads we call the *work threads*. This technique depends on the observation that instructions that do not compute control flow can be deferred with respect to instructions that do. We devise a special architecture to exploit these threads: one processor core is devoted the control thread, and one or more are devoted to the work threads.

The first step to parallelizing a program in this manner is to derive the subprograms that will run on each core. 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. Section 2 introduces this notion of partitioning of a program into the control thread and work threads. This partitioning can be optimized in various ways depending on the available microarchitecture and compiler. The mechanics of carrying out the partitioning are examined by Section 3, which focuses on the selection of the control thread and examines the balance between control and work threads.

Additionally, we need a machine to execute the partitioned program in parallel. The machine proposed in this paper has

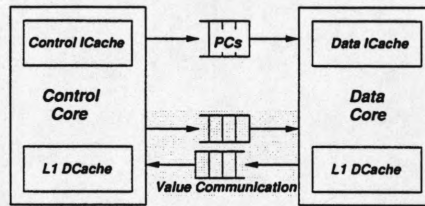


Figure 1. Example of a Control-Data Decoupled Machine

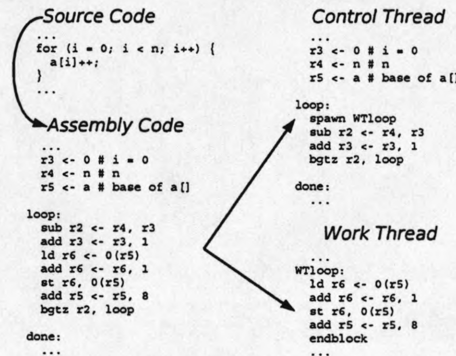


Figure 2. A simple example that illustrates control/data partitioning.

one processing core dedicated to evaluating the control thread, and one or more cores for evaluating the remaining instructions in the work 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. Section 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 work threads. Figure 1 shows a simplified machine model of a control processor connected to one data processor.

We evaluate the performance of our technique on the machine we develop in Section 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 values between the different threads. Finally, it examines some resulting challenges in developing a wider parallel machine.

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

2. DEFINITIONS

This Section defines the *control thread* and *work threads*. Basically, the control thread is a slice of the program that computes the control flow while work threads are spawned the control thread to perform any remaining computation.

Figure 2 provides example code partitioned into control and work threads. In this simple loop, the control thread is highly decoupled from the data computation and hence the work threads. The control thread contains spawn instructions that initiate work threads to perform the data computation within each iteration of the loop.

Below, we define the properties that characterize the control thread and the work threads.

2.1 Properties of the Control Thread

The control thread (CT) is described by a number of properties. A required property for the CT is *control independence*. The CT may further have the property of *completeness* over the control flow or over the data flow. Finally, the CT may have the property of *preciseness*.

The fundamental property a CT must have is *control independence*. In other words, for every instruction in the CT, if that instruction is control-dependent on some control instruction, that control instruction must also be in the CT. This is what allows the CT to run ahead of and make control-flow decisions in advance of where the WT processor is.

A CT is said to have *control-flow completeness* when it contains the entire set of control instructions within the program. When the CT does not have control-flow completeness, it is said to be control-flow incomplete. Provided we maintain control independence, we can break control-flow completeness and move "local" control-flow decisions to a WT. For instance, we may allow the CT to spawn off a WT that executes a loop with a single exit point. The reason for doing this is to reduce the amount of computation done in the CT, to improve its performance.

Likewise, a CT is said to have *data-flow completeness* when all the inputs to all the instructions within the CT are available within the CT, i.e. the CT has no data dependence on any instruction in a WT. When the CT does not have data-flow completeness, it is said to be data-flow incomplete. In the dynamic execution of a data-flow incomplete CT, we will encounter dependencies between a WT and the CT, each of which will require a (potentially costly) communication of data from the WT processor to the CT.

Finally, a CT is said to be *precise* when, for every instruction in the CT, either that instruction is itself a control instruction or there is a data-flow dependence chain from that instruction to some control instruction within the CT. Otherwise, a CT is called *imprecise*, i.e. when it contains some instruction whose outputs are never needed to compute control-flow. For performance, we want to minimize the degree of impreciseness in the dynamic execution of the CT.

Figure 3 illustrates the conflict between data-flow completeness and preciseness when going from the static control thread to a dynamic execution of the control thread. For the given snippet of code, the given static control thread is the minimal set of instructions that maintains data-flow completeness. This includes the instructions computing the value of x. However, in the given dynamic execution of this code, the instructions computing x cause impreciseness; the CT would have more dynamic instructions than needed for the control thread to be dynamically precise. Hence, we may introduce data-flow incompleteness along infrequently executed paths in order to improve preciseness, and hence performance, along the more frequent paths.

2.2 Properties of Work Threads

The work threads must be chosen to maintain the behavior of the program. Hence, any instructions that are not in the CT must be executed in a WT. We call the set of WTs *minimal* if there is no redundancy between the CT and any WT, i.e. no instruction in the original program is placed in both the CT and a WT. Analogous to the CT, we can call a WT *data-flow complete* if it has no data dependences on the CT. Data-flow completeness is far less important for a WT because the CT is running ahead of it and hence any possible inputs are available at the time the WT is spawned. Nonetheless, if data communication from CT to WT is still to expensive, we can put redundant instructions into the WT to eliminate data dependences.

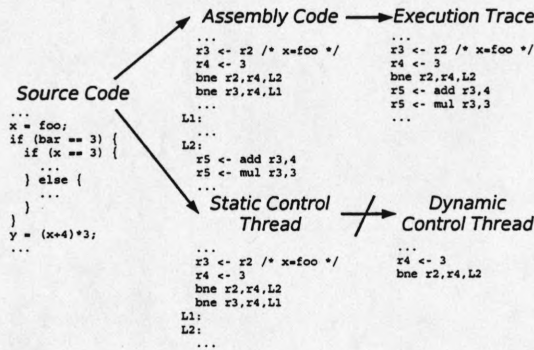


Figure 3. A simple example that illustrates data-flow completeness causing impreciseness.

Clearly, a WT is not control independent of the CT. Similarly, there is little point in making a WT control-flow complete. For the remainder of this paper, we make the set of work threads minimal.

3. COMPUTING THE CONTROL THREAD

As stated above, partitioning a program using our model involves selecting a control thread and work threads from the program code. This Section details a software algorithm for iteratively selecting the control thread. We also detail various optimizations to reduce the size of the control thread and balance the amount of computation between the control and work threads. We describe and evaluate a binary analysis implementation of our algorithm and the proposed optimizations. Finally, we consider alternative methods for selecting the CT.

3.1 Software Iterative CT Selection

We determine which instructions to include in the CT using iterative data-flow analysis. First, we select an initial CT consisting of control instructions. Then, we compute, iteratively, the set of instructions on which the CT control instructions have a data dependence. We describe this process in more detail below.

3.1.1 Initial CT

As noted, the first step in computing a control thread is to generate an initial CT. The initial CT should consist of a set of control instructions that maintains the control independence property as described in the previous section. For now, the initial CT we select is the set of all control instructions, including branches, jumps, and function call/returns. Later, we will consider selecting an initial CT consisting of only a subset of the control instructions. The inititial CT instructions are added to a hash table, the CT_inst_set.

3.1.2 CT Live Variables

The next step in the CT generation is to find the set of variables that are live in the CT at each point in the program. We define a variable as CT_live at a point p in the program if for any possible execution of the program starting at point p, the variable is used by a CT instruction before it is overwritten. We want to add the writers of all CT_live variables to the CT_inst_set.

The computation of CT_live sets is similar to live variable analysis. Like live variables, CT_live variables can be computed by iterative solution of dataflow equations. The dataflow equations for CT_live variables are shown below.

$$CT_live_{in}(B) = CT_use(B) \cup (CT_live_{out}(B) - kill(B))$$

$$CT_live_{out}(B) = \bigcup_{S \in Succ(B)} CT_live_{in}(S)$$

Note that these equations are virtually identical to those for live variable analysis, except for the presence of $CT_use(B)$ in place of $use(B)$. $CT_use(B)$ is the set of variables whose values may be used by an instruction in the CT_inst_set in block B before being defined (not necessarily by a control thread instruction) in B.

The actual computation of the CT_live sets is carried out by iterating over the CFG in reverse order. At the bottom of each basic block, we apply the equations to determine CT_live_{out}. Then, we iterate through the instructions in the block in reverse, keeping track of the CT_live set. If we encounter an instruction that writes a CT_live variable, we add it to the CT_inst_set. We also remove its outputs from and add its inputs to the CT_live set. If we encounter an instruction already in the CT_inst_set, we similarly remove its outputs from and add its inputs to the CT_live set. At the top of the basic block we union CT_live_{in} with the CT_live set from within the block. We do this for every block in the CFG, and then repeat until there are no more additions to either the CT_inst_set or any of the CT_live sets.

3.1.3 Memory Dependences

The CT_liveness algorithm as described above did not specifically consider memory dependences. In order to avoid dependence violations, we need to propagate CT_liveness through memory as well as through register dependencies. This is especially important with stack accesses on x86 (the architecture on which our method is evaluated). Fortunately, it isn't hard to determine memory dependencies through the stack. Determining memory dependencies through the heap, however, requires alias analysis. As we show later on, though, we don't really need to propagate dependencies through the heap in order to get good performance.

3.2 Optimizations to Software CT Selection

The algorithm as described above is effective at computing a CT that is control-flow and data-flow complete (neglecting memory aliasing). However, such a CT is too large for us to get much performance benefit on a multiprocessing machine. To get better performance, we need to reduce the size of the CT to better balance the load between the CT and WT processors.

There are two classes of optimizations that aim to accomplish this. The first class does this by breaking control-flow completeness, moving certain control instructions to a WT while maintaining the control independence property. The second class of optimizations aims to improve the preciseness of the dynamic CT by breaking data-flow completeness. Below, we present local control exclusion (LCE), a subclass of the first class of optimizations, and two profile-directed optimizations from the second class.

3.2.1 Local control exclusion

LCE is a class of CT selection optimizations where we move much of the "local" control flow computation to work threads, while using the control thread for "global" control flow computation. An LCE optimization can be performed over any type of control flow construct that has a single entry and exit point. We apply the LCE optimization by initially excluding the control

instructions from the chosen construct from the initial CT. Then, we perform the *CT_liveness* analysis as usual. Then, we restore the control independence property; if any instruction within the construct is in the *CT_inst_set*, we must add any instructions on which it is control dependent to the *CT_inst_set*, and re-run the *CT_liveness* analysis. We repeatedly iterate between restoring control independence and performing *CT_liveness* analysis until the *CT_inst_set* converges.

One particular LCE optimization that should yield good performance improvements is tight loops exclusion (TLE). TLE is LCE applied over tight loops, that is, loops with single-basic block bodies. The idea is that when we encounter a loop that has nothing to do with determining future control flow beyond the loop, we want to spawn off the loop as a WT and let the CT run further ahead.

3.2.2 Infrequent path pruning

As seen in Figure 3, *CT_liveness* is often propagated across very infrequently or never taken CFG edges. Moreover, the variables that are made *CT_live* by dependencies across infrequent paths can substantially increase the number of instructions that are added to the CT. Because these paths are rarely taken, the aforementioned variables are rarely live in the dynamic CT. In many cases, then, it would be better not to count these variables as *CT_live* at all, in order to improve the preciseness of the dynamic CT, and patch up the infrequent-path dependence violations in hardware.

This exclusion of dependences across infrequent paths is carried out by infrequent path pruning. After selecting the initial control thread, we use edge profiling information to remove infrequently taken edges from the CFG for the *CT_liveness* analysis. The rest of the *CT_liveness* analysis is performed as normally until convergence. Finally, after the CT selection has finished, the pruned edges can be added back in.

If the profile is accurate, the instructions excluded from the CT by the pruning would rarely produce values that are dynamically *CT_live*. Hence, the performance benefit of the improved CT preciseness would outweigh the cost of fixing an occasional dependence violation. Of course, if the profile is not accurate, and the pruned paths are executed more frequently than expected, the dependence violations could severely hurt performance.

3.2.3 Biased branch exclusion

A second profile-directed optimization is biased branch exclusion (BBE). The idea behind BBE is to take advantage of the highly predictable nature of many branches to reduce the size of the CT. In applying BBE, we exclude the inputs of highly biased branches from the *CT_live* set. Thus, a biased branch's inputs would be computed in the WT even if the branch itself were in the CT, because even without knowing the inputs the branch would almost certainly be predicted correctly anyway. Then, once the inputs become available, the branch's direction could be verified, and only if the prediction were wrong would we roll back and re-execute the CT from the point of the branch.

To apply BBE, we first determine from the profile which branches are highly biased, and add these branches to a table. Then, during the *CT_liveness* analysis, we propagate the *CT_liveness* of a variable only if its reader is not one of the biased branches. Finally, we mark each biased branch for delayed resolution.

Like pruning, BBE requires hardware support for data communication, to fetch the biased branches' inputs from the WT. Moreover, BBE also requires the hardware to be able to delay resolution of marked branches until the inputs are available from the WT, rather than when the branch reaches the execute stage of the pipeline. Also like pruning, we can incur a severe performance penalty if the profile is wrong, since now mispredicted branches cause the CT processor's pipeline to be rolled back much further.

3.3 Binary Analysis Implementation

The iterative control thread selection algorithm descibed above was implemented as a binary analysis on x86 code. To do this, we generate a "decoded" x86 binary by running the original instructions through a decoder to generate a list of micro-operations for each architectural instruction. The analyzer computes the CT using the algorithm described above and outputs a text file listing, for each instruction address, whether the instruction belongs in the CT or whether it is in a WT.

The analyzer is able to propagate dependencies across function call/return boundaries. When generating the CFG, each call instruction is treated as a jump to the start of the called function, while each return instruction produces CFG edges to the instruction following each call of that function. Thus, the *CT_live*ness of a register could get propagated from one function f() that calls a particular function g() to any other function that calls g(), provided the register is not killed within function g(). Though this does not hurt correctness, it is non-ideal. However, since x86 has so few registers, it is very likely that most if not all *CT_live* variables are killed within any given function.

The analyzer implements the TLE, infrequent path pruning, and BBE optimizations. Because our implementation of BBE requires pruning to identify the frequent direction of biased branches, we can only run BBE when we also run pruning. For pruning and BBE, we generate edge profile information by running a trace of each program through a functional simulator.

Because dependencies propagated through the stack are easy to identify, we are able to handle them perfectly. Other memory dependencies can only be captured through simulation however. Thus, by default we make no use of non-stack memory aliasing information, although we can generate perfect aliasing information in the same simulation run we use to generate the edge profiles.

The analyzer's partitioning was tested on 7 benchmarks from SPECint2000 and 7 benchmarks from Winstone, shown in Table 1. To determine the dynamic size of the CT and WT, we used execution traces of 50 million instructions for each SPECint benchmark and 100 million instructions for each Winstone benchmark. The choice of benchmarks was driven by the availability of these instruction traces.

As can be seen in Figure 4, the static CT is a large fraction of the program. Moreover, TLE does very little to reduce the static size of the CT. Of course, because the few instructions removed are within loops, the dynamic size of the CT is reduced by much more. BBE and pruning, on the other hand, considerably reduce the static size of the CT.

Figure 5 shows the dynamic size of CT. On SPEC, with the perfect aliasing information, the CT size 94% as large as the original instruction stream (97% when including the spawns). This falls to 76% (85%) percent when the use of aliasing is turned off, to 72% (79%) with TLE, and to 74% (83%) with pruning. BBE again led to the largest reduction in CT size, however, and with all 3 optimizations combined, the CT was just 53% of the size of the original instruction stream (65% including spawns). Keep in mind, though, that pruning and BBE also introduce potential dependences from WTs to the CT.

3.4 Alternative CT Selection Methods

As an alternative to the method described above, the partitioning could be performed dynamically, in hardware. 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. Moreover, the profile-guided optimizations would be more effective if they could take advantage of a program's phased behavior and hence further improve the preciseness of the selected CT. 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

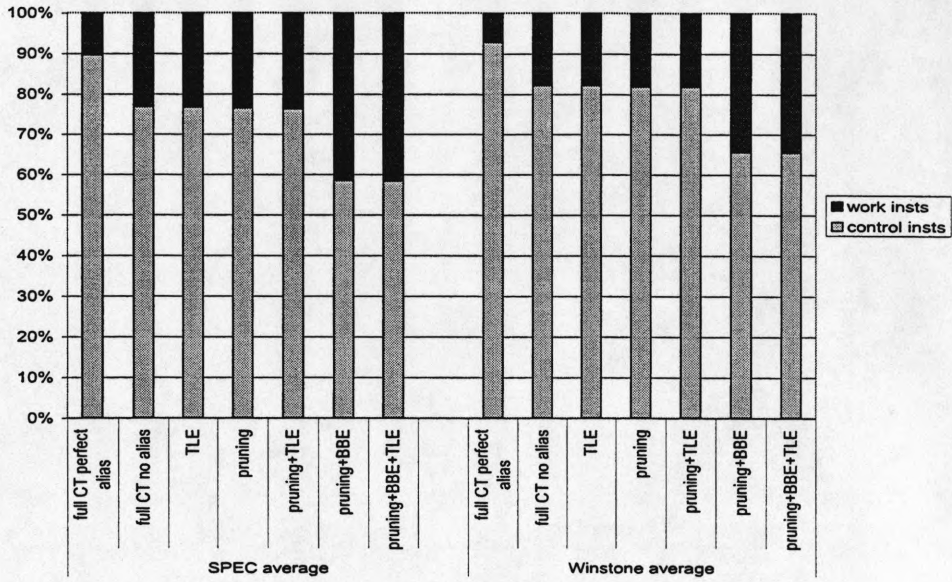


Figure 4. Static size of control thread.

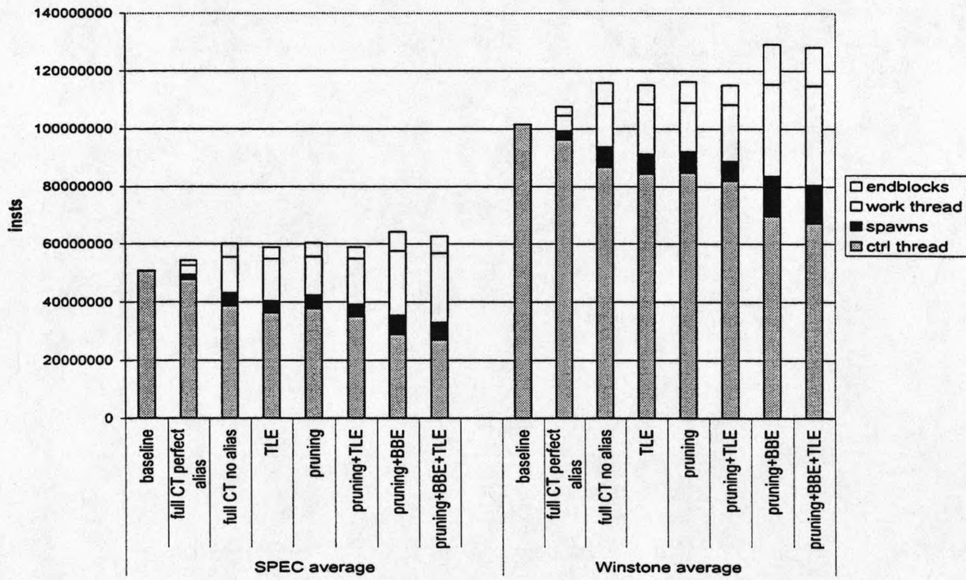


Figure 5. Dynamic size of control thread.

Table 1. **Benchmarks used for evaluating the control/data partitioning**

benchmark	suite	size (dynamic insts)
bzip2	SPECint2000	50M
crafty	SPECint2000	50M
eon	SPECint2000	50M
gzip	SPECint2000	50M
parser	SPECint2000	50M
twolf	SPECint2000	50M
vortex	SPECint2000	55M
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

with a program optimized to use the resources of only one. The details of control thread selection in hardware are beyond the scope of this paper.

4. PROPOSED ARCHITECTURE

This Section describes the proposed architecture to perform parallel execution of the control and work threads. To the instruction set of this architecture, we add new instructions to support efficient spawning and ending of work threads. In the hardware, we add hardware to support the spawning of work threads, to support register communication between control and work threads, and to check for and fix memory dependence violations between control and work threads.

4.1 Instruction Set Architecture

The instruction set architecture must be extended in order to manage the control and work threads. Specifically, there needs to be a way for the CT to efficiently spawn work threads, and for the WT blocks to mark their endpoints.

We add two new instructions to the ISA (x86 in this case). Spawns are handled by the *pbr* (parallel branch) instruction. The instruction contains the opcode (1 byte) and a 16-bit immediate providing an offset to the location of the spawn point in the work thread. Endblocks are handled by the *pjn* (parallel join) instruction, which consists of a 1-byte opcode.

Semantically, a *pbr/pjn* pair acts similarly to a call/ret pair, with the logical program order flowing from the *pbr* to the spawn point to the *pjn* and back to the instruction following the spawn point. In actual execution, the machine attempts to execute the instructions following the spawn point in parallel with the instructions following the *pbr*, with the hardware guaranteeing that dataflow dependences are not violated.

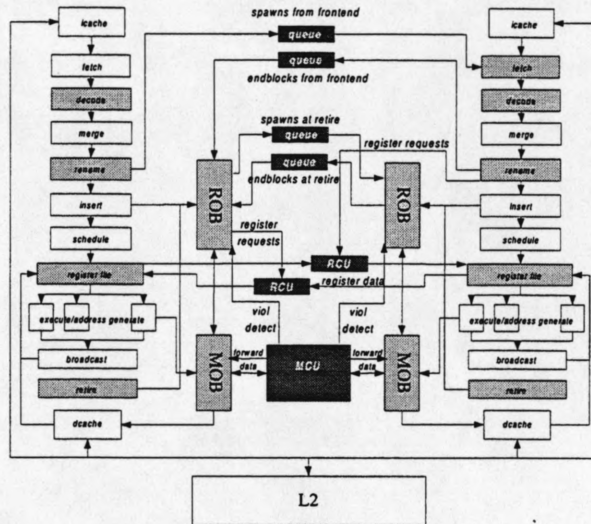


Figure 6. Our CMP architecture, with added hardware for CT/WT parallelism shown in dark gray.

4.2 Microrchitecture

Our hardware microarchitecture, shown in Figure 6 is based on a dual-core CMP. Each core is a deeply-pipelined, out-of-order processor, with a microarchitecture somewhat similar to the AMD K8. The CMP is augmented with additional hardware (shown in dark gray) to implement thread spawning, register coherence, and memory coherence.

4.2.1 Instruction Fetch

In our execution model, the processor core executing the CT drives the fetch order of the WT core. When the CT core decodes a *pbr* instruction, it sends the target of the spawn to the WT core. The WT core buffers all such spawn requests in a FIFO. At the fetch stage, the WT core initiates fetch from the location of the spawn. If the WT core encounters a *pjn*, it initiates fetch from the next spawn in the FIFO, or stalls if no spawns are enqueued.

In actual implementation, the WT core treats *pjn* instructions as control instructions, and uses its branch predictor/BTB to predict where the next spawn point will be and begin fetching from there immediately. Instructions following the *pjn* are then stalled at the decode stage until the next spawn enters the queue. Thus, the hardware treats each *pjn* as a branch, and a dequeue of the next spawn as the "resolution" of the "branch".

One complication arises from branch mispredictions in the CT. 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 WT core to squash its corresponding, incorrect speculative instructions. This requires that a WT not begin to retire until the *pbr* that spawned it also retires; hence the need for a second spawn queue in the retire stage.

4.2.2 Register Communication

All register communication between the work and control threads can be identified explicitly. Mechanisms for handling register communication were developed in [11] and [7]. We propose a different mechanism that is better suited to our

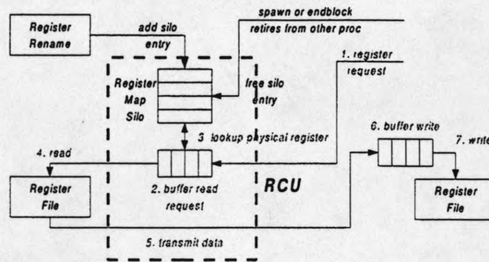


Figure 7. Register communication unit for one of the processor cores. Both cores have their own RCU.

partitioning approach.

At the rename stage, both processors maintain a register update bitmask recording the set of architectural registers that are being updated by the current group of instructions. The CT processor, when it encounters a *pbr* instruction, sends the bitmaskmask to the work thread processor in the spawn request along with the next fetch PC, and resets the bitmask. Likewise, the WT processor, when it encounters a *pjn* instruction, sends and resets its register update bitmask back to the ROB in the CT processor. In our x86 implementation, the register update mask is 108 bits wide, as it must support partial register writes, the flags register, and microcode temporaries.

When the WT processor dequeues a spawn from the queue, it reads the bitmask to discover the set of registers that have been written in the CT. After the decode stage, the WT processor can identify which instructions' input registers are valid and which need to be fetched from the CT. In the rename stage, new registers are allocated for these inputs and a request for the registers in the CT is enqueued, as shown in Figure 6.

As stated, when the WT processor encounters a *pjn* instruction, it sends its register update mask to the CT processor, along with an identifier for the originating spawn instruction. The CT processor updates its register file by reading all the new values produced by the WT. To minimize communication between the processors these updates are queued up for several blocks before being carried out. The CT processor also checks before the retire stage to see whether instructions in the CT incorrectly read any stale values in the updated registers; if so, it enqueues a request for these registers. When the request is completed it replays any dependent instructions.

The actual register communication is carried out by the structure shown in 7. The mechanism contains, on each processor's side, a register map silo, a buffer of pending register requests originating on that processor, and a buffer of pending reads from that processor's register file. The register silo contains mappings from the architectural register to the physical register at the point of each *pbr* or *pjn* instruction that remains in the system. Each register request contains an identifier for the *pbr* or *pjn* instruction preceeding it, the architectural register being requested, and the physical register into which the result is to be written.

When a register request arrives from one processor, it is sent to the other processor and buffered. At the other processor, the physical register corresponding to the requested architectural register is looked up, and when that physical register becomes ready the value is read (on a dedicated read/write port on the processor's register file) and sent back to the originating processor. The originating processor then writes the value into its register file; the write signals instructions waiting for that register to wake up.

Instructions in the WT cannot retire until the corresponding *pbr* in the CT retires. Likewise, instructions in the CT

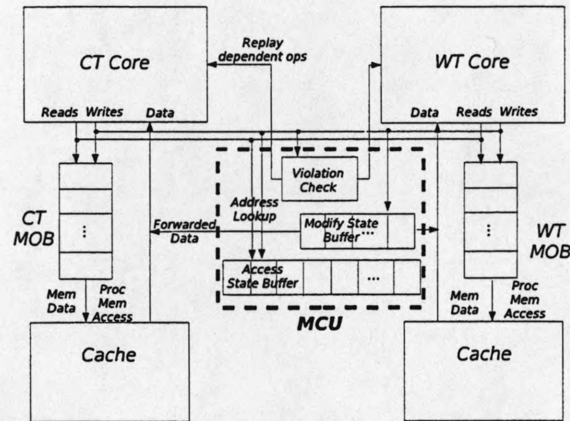


Figure 8. Memory communication unit.

following a *pbr* cannot retire until we can guarantee that there are no more spawned instructions in the WT that may produce a value needed by the following CT instructions, in other words until the corresponding *pjn* instruction has been decoded in the WT.

Hence, the CT processor, in addition to sending spawns at the decode stage, needs to send a retirement notification for those spawns in the retire stage. Similarly, the WT processor needs to send an endblock notification at both the decode and retirement stages.

The policy for allocation and deallocation of registers must change to accomodate register communication. In the case where a register is written and then overwritten without any intervening *pbr* or *pjn*, the register can be deallocated when the overwriting instruction commits, as usual. However, if there is an intervening *pbr* or *pjn*, the register cannot be deallocated until the prior block has retired on the other processor.

This register communication mechanism requires the compiler computing the CT and WTs to guarantee that register values generated by the WT do not get used by the CT for at least an instruction window's number of instructions. Otherwise, the CT processor will repeatedly need to replay instructions, hurting performance.

4.2.3 Memory Communication

As with register values, the hardware needs to guarantee that memory data is communicated correctly. Correct, high-performance execution requires efficient communication of memory dependences, timely detection of incorrect data loads, as well as quick recovery in case of such violations.

A mechanism for detecting memory dependence violations is proposed by [7]. 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 re-execute the threads from the correct point.

To detect data dependence violations and to communicate data values, the control/data processor uses the memory communication unit shown in Figure 8. The structure consists of two tables, the modify state buffer and the access state buffer, and some logic that examines these tables for dependence violations. The MSB stores the logical instruction sequence number, memory address, and data value for each store. The ASB stores the logical instruction sequence number and memory

address for every load. Both the MSB and the ASB are CAM structures, with the memory address used as the index.

When a load is inserted into the pipline, it is inserted into the ASB along with a unique identifier that can be used to determine the logical ordering of all memory operations, effectively a sequence number local to the MCU. When the address for the load is generated, the address is also put in the ASB. The memory communication unit 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 inserted into the pipeline, it is assigned a sequence number and inserted into the ASB as well as the MSB. When the address for the store is generated and the value becomes available, a lookup is perfomed in the MSB for all other stores with the same address. If any such stores are found, the one with the smallest sequence number greater than that of the current store is selected. A lookup is then performed in the ASB for all loads with the same memory address and sequence numbers in between the current store and the store selected from the MSB. If any such loads are found, there has been a dependence violation. If a memory dependence violation is detected, we must be able to replay any loads that returned incorrect data.

In order to be able to recover state at 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 CT can commit only when all logically earlier load or store operations in the WT have also been committed.

The mechanism for committing memory operations in order is also part of the memory communication unit (not shown in Figure 8). The ASB keeps a register indicating which processor core initiated the oldest non-committed memory access within the ASB. The processor cores check this register when they attempt to commit a memory operation. Only the core indicated by the register can commit memory operations.

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

5. PERFORMANCE EVALUATION

In this section we evaluate the performance of the above architecture and partitioning schemes using a microarchitectural simulation. We find that we can acheive average speedups of 1.14 on SPEC and 1.17 on Winstone. We find that these speedups are insensitive to interprocessor communication and latency. Finally, we find that the speedup of our parallel method is scalable as superscalar machines grow larger.

5.1 Methodology

We simulate the performance of the processor proposed in Section 4 on the benchmarks and partitioning heuristics from Section 3. We use a detailed x86-based microarchitecture simulator that implements spawn/endblock synchronization, memory communication, register communication, and stalls due to data sharing violations.

5.1.1 Simulation Environment

The environment used to evaluate the performance of control/data partitioning is Joshua, a multiprocessor microarchitectural simulator. Joshua implements the x86 ISA by executing a set of RISC-like micro-operations derived from decoded x86

Table 2. Simulated realistic processor parameters (per processor core)

processor parameter	value
Issue/retire width	4
In-flight instructions	256
Integer units	8: 3 simple int, 1 complex int, 1 multiply, 2 load, 1 store
FP units	3: 2 simple FP, 1 complex FP
Pipeline latency	minimum 15 cycles
L1 Cache	8KB/1 cycle latency instruction and 64KB/2 cycle latency data
L2 Cache	1MB, 9 cycle latency
Memory	199 cycle latency
Branch prediction	2K entry BTB plus hybrid predictor

Table 3. Simulated dual core additional parameters

CMP parameter	value
Spawn queue size	256 spawns
Register communication queue	128 pending reads per processor
Register communication latency	2 cycles minimum
Register communication bandwidth	2 communications per cycle
ASB size	320 memory operations with 128 reserved per processor
MSB size	160 stores with 32 entries reserved per processor
Memory bypass latency	5 cycles minimum
Memory bypass bandwidth	1 bypass per cycle
Cache coherence	update protocol

instructions. Joshua is trace-driven, and it can execute from the x86 trace directly or it can execute a pre-decoded trace of uops. Joshua includes a module for simulating a CMP with multiple execution cores; one core is used to simulate the control thread and the other the work thread.

The spawn queues, endblock queues, memory sharing mechanism, and register sharing mechanism, as described in Section 4, are implemented as modules in Joshua.

Table 2 gives the parameters for the baseline uniprocessor. Table 3 gives additional parameters for the corresponding control/data parallel CMP. The baseline represents an aggressive but realistic next-generation superscalar processor, with 4-wide sustained execution and a 15 stage integer pipeline. Table 4 gives the parameters for a doubled superscalar processor, with twice the execution bandwidth, instruction window, and memory ports. The doubled processor represents the limit of what is feasible in a conventional superscalar processor.

5.1.2 Benchmarks

The performance was 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 run in a "hot-spot" trace of 50 to 100

Table 4. Simulated 8-wide processor parameters (per processor core)

processor parameter	value
Issue/retire width	8
In-flight instructions	512
Integer units	16: 6 simple int, 2 complex int, 2 multiply, 4 load, 2 store
FP units	4: 3 simple FP, 1 complex FP
Pipeline latency	minimum 15 cycles
Memory system	same as 4-wide
Branch prediction	1K entry BTB plus gshare with 18 bit history

million x86 instructions selected as being representative of the execution of the program. Table 1 in Section 3 lists the benchmarks.

5.2 Results

The performance of the dual 4-wide machine running in parallel is compared with the conventional 4-wide and 8-wide superscalars in Figure 9. In 3 of the SPEC and 5 of the Winstone benchmarks, the parallel 4-wide machines outperform the 8-wide machine, with the parallel machines achieving (with the best selection algorithm) an average 1.14 speedup on SPEC and a 1.17 speedup on Winstone, while the 8-wide superscalar achieves an average 1.13 speedup on SPEC and an average 1.12 speedup on Winstone. On most benchmarks, the best performance is achieved by the TLE optimization by itself, though in several cases other optimizations helped, and in the case of bzip2 CT selection with no optimizations performed best.

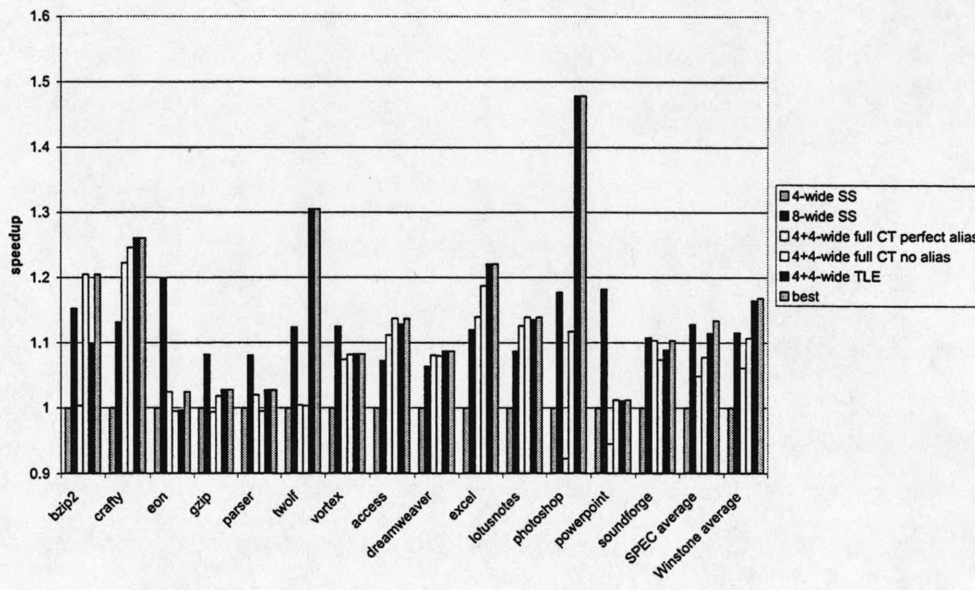
5.3 Analysis

We examine the performance effect of the availability of alias information on the performance of the parallel architecture. We further examine the effect of the various proposed optimizations. We also evaluate the sensitivity of the parallel machine's performance to the latency and bandwidth of interprocessor communication.

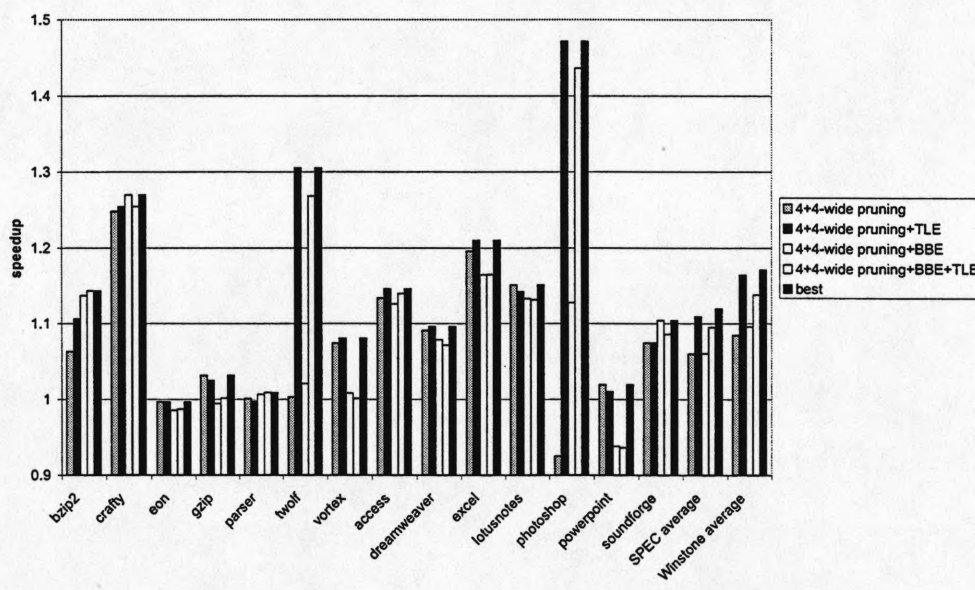
5.3.1 Effect of alias information

The effect of memory aliasing information is mixed. Overall performance of the unoptimized control thread is sometimes better with perfect aliasing information, as in the case of parser, twolf, and soundforge. Much of the time, performance is the same either way, as with vortex, dreamweaver, and lotusnotes. Other times, the performance is better with no aliasing information, as in bzip2, crafty, eon, gzip, access, excel, photoshop, and powerpoint; with bzip2 and photoshop it is dramatically better.

The reason why parser, twolf, and soundforge perform better when alias information is available is straightforward: you avoid memory dependence violations. More interesting is why some benchmarks perform better without the information. The basic reason is that including the (non-stack) memory dependences in the control thread selection causes more instructions to be added to the control thread. In many cases, however, the aliasing is of a may-alias type, so that much of the time the actual memory dependence may only occur rarely in the dynamic instruction stream. Thus, most of the time including the memory dependence in the control thread selection simply causes more instructions to be added to the control thread than necessary.



(a)



(b)

Figure 9. Speedup of the control/data parallel machine compared with the 4-wide and 8-wide superscalars. (a) selection algorithms without profiling (b) selection algorithms with profiling.

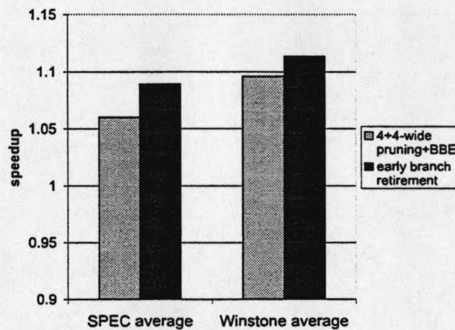


Figure 10. Effect of eliminating register dependence checks before CT resolves biased branches.

The most extreme case of such may-alias information is when the dependence is in a loop that iterates over an array. In this case, the alias information will indicate a dependence between any pair of loads and stores over this array, whereas in fact such loads and stores may rarely alias in the dynamic instruction stream. Notably, both benchmarks (bzip2 and photoshop) that performed much better without alias information spent a lot of time in compact loops iterating over large arrays.

5.3.2 Effect of TLE

TLE provides a solid performance gain on most benchmarks. By moving loops that aren't important to control flow off to a WT, we allow the CT processor to slip considerably further ahead of the WT processor. Moreover, whereas before we would have one spawn in each iteration of a tight loop, we now have a single spawn instruction for the whole loop. This reduction in the number of spawns, which is clearly seen in Figure 5 from Section 3, reduces the effective size of the CT, and more importantly reduces the amount of syncronization between the CT and WT processors.

In most benchmarks, TLE alone provides the best performance. In a few cases, such as access, dreamweaver, and lotusnotes, it is TLE+pruning. In the case of crafty and soundforge, TLE performance is about the same as without TLE. Only bzip2 does worse with TLE than without. In fact, in bzip2 TLE moves enough code to the WTs that the WT processor often becomes the bottleneck; the workload in bzip2 is better balanced when no optimizations are applied.

5.3.3 Effect of pruning and BBE

As seen in Figure 9b, pruning infrequent paths hurts performance slightly when compared to the full CT (both case without aliasing info). In fact, in some benchmarks it helps performance and in others it hurts it. Pruning introduces register dependencies from the WT to the CT on those infrequent paths, and these can be very costly, especially when the WT is running well behind the CT. Most likely, this optimization requires additional tuning to make this case more rare.

BBE offers some performance benefit over just pruning. As seen earlier, BBE is a very powerful optimization in reducing the control thread size. The performance of BBE+pruning, however, is still slightly worse than the full CT. The reason for the performance loss is that BBE introduces yet more register dependencies. In this case, the register dependencies require biased branches in the CT to remain in the ROB until the WT catches up, thus removing the advantage of allowing the CT to run further ahead. If we eliminate the requirement that correctly predicted biased branches remain in the machine until the inputs are ready (which is unrealistic), we can see a clear performance benefit, as shown in Figure 10.

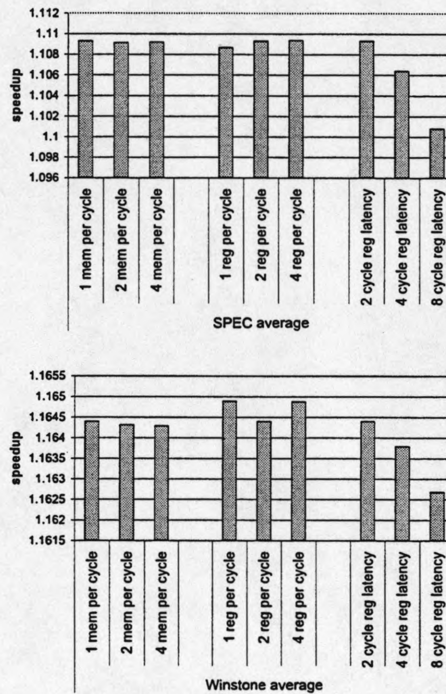


Figure 11. Speedup of the control/data parallel machine when varying register communication bandwidth, memory communication bandwidth, and register communication latency.

5.3.4 Effect of communication bandwidth and latency

Figure 11 shows the effect of varying memory and register communication bandwidth and register communication latency. The effect of varying the memory communication bandwidth is negligible. Even the ability to bypass one memory value per cycle through the MCU is more than sufficient. The effect of varying the register communication bandwidth is also very small. In SPEC, there is an improvement in eon as the bandwidth increases, which accounts for the overall improvement. In Winstone, there is a very small but surprising drop in performance with 2 register communications per cycle (the default). This is due solely to a small (0.3%) performance drop in access. Increasing the register communication latency has a slightly bigger effect on performance. This is due primarily to performance drops in eon (5.4%) and powerpoint (1.0%), which are among the worst performing benchmarks even when the latency is small. Overall, though, once communication bandwidth is large enough and latency small enough, the performance is unaffected by further improvement.

5.3.5 Slip between WT and CT

Fundamentally, the performance benefit of parallel execution comes from being able to reorder the instruction stream to find independent instructions. Figure 12 shows the distribution of the number of cycles the WT core lags the CT core with TLE. It dramatically illustrates the effect of greater reordering of the instruction stream, with the 7 benchmarks with higher speedup having much greater slip between the CT and WT than the 7 benchmarks with lower speedup.

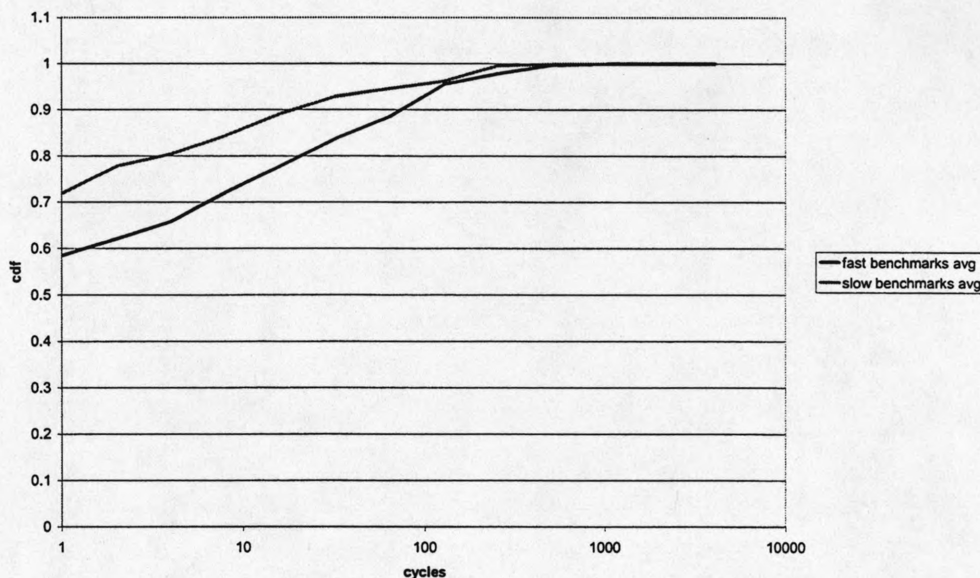


Figure 12. Delay between CT enqueuing a spawn and WT beginning fetch at the spawn point with TLE.

5.3.6 Scalability of control/data parallelism

We examine the scalability of the control/data parallel paradigm in Figure 13, which shows the performance of the dual-core machine as WT core, the CT core, and both cores are doubled in size. The speedup goes to 1.34 for SPEC and 1.35 for Winstone with the dual 8-wide machines. In fact, SPEC and Winstone with dual 8-wides both achieve a speedup of 1.2 relative to the single 8-wide superscalar. This suggests that the parallel machine actually takes better advantage of the expanding resources than a single-core machine. In particular, as mentioned at the end of Section 4, the increased ROB size greatly alleviates the impact of the retirement constraints on memory operations.

5.4 Limitations

The accuracy of these results is limited by the limitations of the simulator infrastructure being used. The version of Joshua being used has several limitations worthy of note.

First, our simulator does not model wrong-path execution. Though this is considered a second order effect in uniprocessor performance, wrong path execution can have an effect in terms of warming up caches that we do not capture. Indeed, this "warm-up" effect constitutes a major portion of the performance benefit in speculative multithreading [1].

Second, our simulator does not implement the complete x86 ISA, as many instructions in the ISA are obscure and complex to implement. In particular, we do not handle many of the x87 FPU state manipulation, BCD arithmetic, SIMD (i.e. MMX, SSE), or priveleged OS instructions. In practice, these unsupported instructions constitute about one out of every thousand instructions. When we encounter any of these instructions, we serialize the machine. We allow all previous instructions to complete execution and retire, and then copy the architectural state from the trace into the machine before resuming execution.

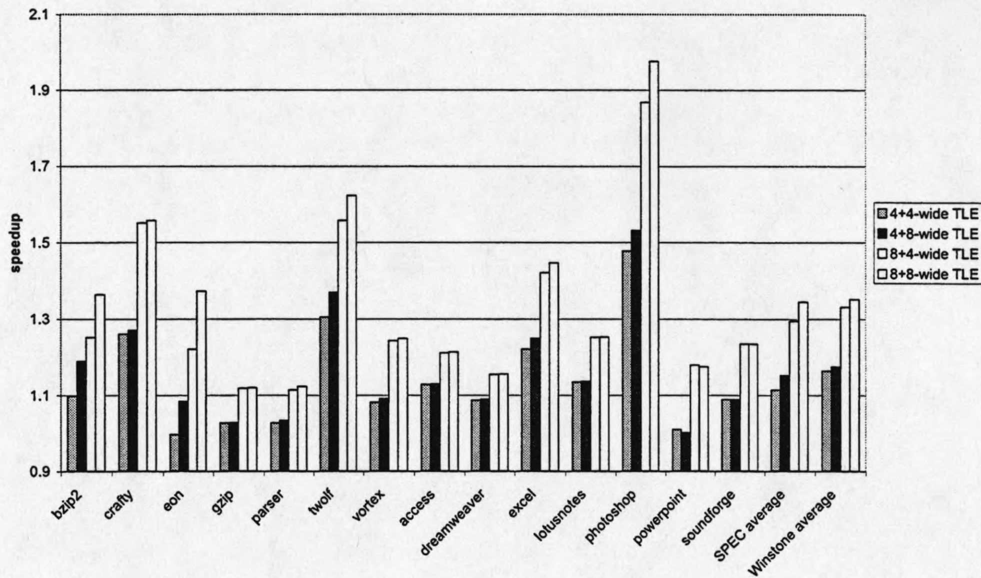


Figure 13. Speedup of control/data processor with either the WT core, the CT core, or both cores expanded to the 8-wide configuration.

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 multi-threaded 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 [10]. 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.

Another early attempt at speculative parallelization is the Multiscalar processor [5] [11]. 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 [9], [7] and [12], propose mechanisms that require less hardware overhead. In [9], Olukotun et. al. describe how to support fine-grain speculative parallelism on the Hydra CMP. In [7], Krishnan and Torrellas propose a CMP design that adds a modest set of structures for speculative multithreading, specifi-

cally 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. The architecture is further developed in [2]. In [12] and [13], Steffan et. al. describe an approach that imposes even less hardware overhead, by extending cache coherence mechanisms to detect dependence violations.

Whereas the preceeding 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 [14] 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.

The SCALE Vector-Thread architecture proposed in [6] is also an approach to parallel architecture. Like our control/data parallel system, SCALE contains a control processor that enqueues execution blocks for other processing cores in the system. SCALE uses a different algorithm for parallelization than us, and it has a different, vector architecture that is oriented towards numerical or embedded processing.

A program-slicing type approach is taken by various work done on prefetching helper threads [3] [8] [4]. In these approaches, a helper thread is constructed from the program slice used to compute memory addresses rather than control-flow. The helper thread runs concurrently with the main thread in order to reduce the cache miss rate of the main thread.

7. Conclusion and Future Work

Automatic parallelization of sequential code is a hard problem. In this paper, we present the control/data parallel model, a novel technique for breaking sequential code into threads.

We extract from the program a control thread, which is a slice of the program that computes control flow, and move the rest of the computation to work threads that can be spawned by the control thread. We present an algorithm to select the control thread in software, and we present several optimizations that break the control thread's completeness properties to improve performance. We also develop an architecture based on a dual-core chip-multiprocessor that can efficiently execute these threads in parallel.

We evaluated the performance of our parallel architecture on benchmarks from SPEC 2000 and Winstone. We achieved speedups, in execution cycles, ranging from 3% to 48% over a conventional 4-wide superscalar x86 processor, averaging 16%. We found that the TLE optimization performed best overall, while profile-directed optimizations only helped performance sometimes. Further, we found that the speedups increase substantially as the size of each core is increased, as we achieve a 20% average speedup with TLE over an 8-wide superscalar.

As future work, we plan to examine a wider set of optimizations to the control thread selection. In particular, in this paper, we examine local control-flow exclusion on only one control-flow construct, that of tight loops. However, LCE optimizations can be applied to any control-flow construct with a single entry and exit point, and hence may offer substantial opportunities to further reduce the size of the control thread. In addition, we plan to examine the possibility of extending the control/data parallel model both to more specialized architectures and to architectures with more than 2-way parallelism.

8 Acknowledgments

We thank the other members of the Advanced Computing Systems group. In particular, we want to thank Greg Muthler, Brian Slechta, Francesco Spadini, and Brian Fahs, who assisted in developing the Joshua simulation infrastructure. We would also like to acknowledge AMD's contribution of execution traces.

References

[1] Y. Chen, R. Sendag, and D. Lilja. Using incorrect speculation to prefetch data in a concurrent multithreaded processor, 2002.

[2] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24, Vancouver, Canada, June 2000.

[3] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. fong Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. pages 14–25.

[4] P. W. et. al. Helper threads via virtual multithreading on an experimental itanium 2 processor-based platform. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[5] M. Franklin. The multiscalar architecture. Technical Report 1196, Computer Sciences Department, University of Wisconsin - Madison, Nov. 1993.

[6] R. Krashinsky, C. Batten, S. Gerding, M. Hampton, B. Pharris, J. Casper, and K. Asanović. The vector-thread architecture. In *31st International Symposium on Computer Architecture*, Munich, Germany, June 2004.

[7] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, Sept. 1999.

[8] C. Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. pages 40–51.

[9] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *International Conference on Supercomputing*, pages 21–30, 1999.

[10] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.

[11] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[12] 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.

[13] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, pages 65–, 2002.

[14] 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.

[15] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002.