

# Hardware Support for Software Controlled Multithreading

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

## Abstract

*Chip multi-processors have emerged as one of the most effective uses of the huge number of transistors available today and in the future, but questions remain as to the best way to leverage CMPs to accelerate single threaded applications. Previous approaches rely on significant speculation to accomplish this goal. Our proposal, NXA, is less speculative than previous proposals, relying heavily on software to guarantee thread correctness, though still allowing parallelism in the presence of ambiguous dependencies. It divides a single thread of execution into multiple using the master-worker paradigm where some set of master threads execute code that spawns tasks for other, worker threads. The master threads generally consist of performance critical instructions that can prefetch data, compute critical control decisions, or compute performance critical dataflow slices. This prevents non-critical instructions from competing with critical instructions for processor resources, allowing the critical thread (and thus the workload) to complete faster. Empirical results from performance simulation show a 20% improvement in performance on a 2-way CMP machine, demonstrating that software controlled multithreading can indeed provide a benefit in the presence of hardware support.*

## 1. INTRODUCTION

As superscalar processors approach practical limits to their complexity, the industry is shifting toward chip multiprocessors (CMPs) to maintain efficient use of available transistors. CMPs efficiently increase throughput for multiple, concurrent applications or multi-threaded applications. Unfortunately, they do not provide the performance speedups for single-threaded applications that the computing community has grown accustomed to.

It is possible to convert single-threaded applications to a multi-threaded target by hand or by compiler. However, the manual process is tedious and compiler techniques are ineffective on general code. Moreover, even when code can be parallelized, a CMP cannot take advantage of any but the coarsest grained parallelism, limiting the performance potential.

To alleviate these limitations, we propose NXA, a software/architecture approach which utilizes static, compile-time analyses to partition workloads into master and worker

threads. Master threads generally consist of performance critical instructions and are given the full resources of a single processor. Worker threads are spawned off from master threads onto other processors. This prevents worker thread instructions from competing with critical master instructions for processor resources, allowing the master threads (and thus the workload) to execute faster. Further details on high level NXA concepts are provided in Section 2.

We present an implementation of NXA to support master-worker threading all the way down to a fine granularity. This implementation of NXA builds on a conventional CMP by adding hardware mechanisms for spawning out worker threads from one processor core to the others, as well as mechanisms to maintain a consistent memory and register state across the cores. This implementation of NXA is described in detail in Section 3.

To create master threads from a workload, we propose and evaluate algorithms to identify critical portions of the workload. Since different workloads present different critical paths to the hardware, we develop three slicing methods to find the critical portions. In the first scheme, control decoupling, the main thread computes the global control flow through the program while the work threads perform data computation. In the second scheme, memory decoupling, the main thread advances miss-prone loads while the work threads defer computation dependent on those loads. In the third scheme, critical path decoupling, the main thread computes values along the dataflow critical path while the work threads defer non-critical computation. Each of these decoupling schemes is described in Section 4.

We evaluate the performance improvement offered by NXA and observe an average speedup of 20%. As expected, we find that each decoupling system works best on certain classes of workloads. The performance of NXA and the decoupling schemes are examined in Section 5.

Section 6 compares and contrasts prior work in decoupled architecture. Finally, Section 7 provides concluding remarks.

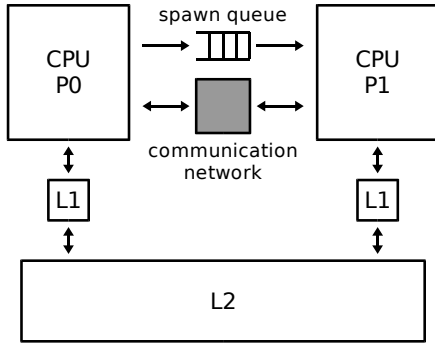


Figure 1. NXA conceptual model.

## 2. PROGRAMMING MODEL

In this section we introduce the NXA programming model. We describe the NXA architecture at a conceptual level as a chip multiprocessor with a mechanism to queue thread spawns between cores and a communication network to maintain a consistent register and memory state. We introduce the decoupling model used to break up single-threaded programs into threads to run on NXA. Finally, we define NXA as a compiler target by examining the interface between the compiler and the NXA hardware.

### 2.1. Conceptual Architecture

A conceptual model of NXA is shown in Figure 2.1. The architecture contains two processor cores, P0 and P1. Connecting the processor cores is a spawn queue, which allows P0 to offload work to the other processor core. When the code running on P0 comes upon a spawn instruction, the target of the spawn is pushed onto the spawn queue. P1 then dequeues the spawn from the queue and executes beginning at the target location. Hence, the main thread running on P0 can run work threads on P1 without the high overhead of forking a full-blown thread.

Also connecting the execution cores is a communication network. The purpose of the communication network is to ensure that instructions on P0 and P1 both see a well defined execution state. In particular, data that is generated in the main thread before a spawn is enqueued needs to be visible in the work thread. Hence, the communication network must be able to grab data from P0 and transmit it to P1 on request. Moreover, the main thread may issue a spawn without knowing for certain that it doesn't have a data dependence on the spawned work thread. In this case, the P0 must signal the communication network that it needs to see the data generated by the work thread on P1. The communication network can stall the main thread, forward any data that may have been incorrectly read by the main thread, or allow the main thread to move forward once it has proved that there are no dependence violations. All this communication is transparent to the software, which only sees a continuous flow of architectural state that remains the same as it would if the program were all run on a uniprocessor.

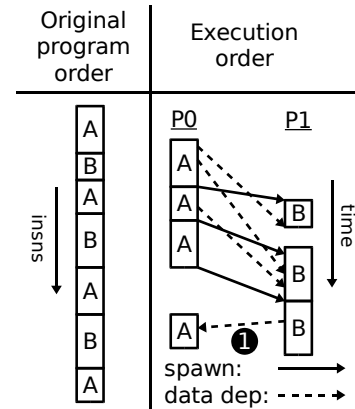


Figure 2. NXA decoupling model example.

Finally, the cores are connected via the memory system. In NXA, the caches for the two cores are connected by a cache coherence system, in particular cache coherence with an update protocol, which is well suited to the producer-consumer relationship between the main and worker threads.

Compare NXA with a conventional CMP. In a conventional CMP, threads are initiated via OS support, by a system call that creates a new process control block, explicitly copies over the initiating thread's state, and assigns to a new processor via the OS kernel's thread scheduling system. Such a heavyweight system for thread creation makes it impossible to gain performance from any but the coarsest grained parallelism, a restriction which excludes much of the program's intrinsic parallelism. Moreover, the conventional CMP has no communication network directly linking the processing cores. All data communication must flow through memory. This severely limits the compiler's ability to extract thread-level parallelism, as it must carefully synchronize inter-thread communication via shared memory and using synchronization calls provided by the OS. Experience has shown that conventional CMPs are a difficult target for automatic parallelization for most workloads, and NXA aims to alleviate the limitations.

### 2.2. Decoupling Model

NXA offers a platform for any arbitrary decoupling scheme. However, in this work we evaluate NXA as a platform for implementing a slicing approach to parallelization. The basic idea of the slicing approach is to find parallel threads that can be interleaved in the original program; each thread is a chain of dependent instructions that constitutes a "slice" of the program. Figure 2.2 provides a graphical reference for the ideas discussed in this section.

Slicing parallelization is implemented on NXA by decomposing the original program into a main thread and worker threads (labeled A and B respectively in the example). Some slice of the program is extracted from the original code and used to form the main thread. This main slice consists of the portion of the code that is critical in determining the program's execution time. For instance, this crit-

ical set may be computing the control flow through the program, or it may be servicing cache misses. In any case, the code within this slice must run as quickly as possible. Code outside of this slice is placed into work threads. This code should be non-critical to the performance. It should consist of instructions or tasks that can be deferred without penalizing the execution time.

On NXA, the main thread runs on P0. With the program's non-critical instructions deferred to work threads, we reduce the resource constraints impeding the performance of the critical instructions in the main thread. When the main thread arrives at a point in the original program where a work thread task needs to be run, it issues a spawn instruction (solid arrows in Figure 2.2) instructing P1 to begin running that task. The communication network provided by the NXA hardware transparently forwards any data needed by the work thread from P0 to P1.

In all slicing approaches to parallelization, the flow of data values (dotted arrows in the figure) and control decisions is predominantly from one slice to another, with little or no flow in the reverse direction. On NXA, the main thread is the slice of the program running out in front, while the work thread is lagging behind. Hence, the bulk of the communication between the main and work threads must be from the main thread to the work thread. Any communication in the reverse direction risks a data dependence violation (data dependence "1" Figure 2.2). Although the communication network is capable of fixing these violations, they are costly for performance. The main thread must be mostly data independent of the work thread for good performance. A similar limitation applies to control dependences.

### 2.3. Compiler Target

Our goal is to provide a target for automatic, static parallelization of code by a binary analyzer or compiler. To this end, NXA exposes three new instructions to the software for spawning work threads and ending execution on a thread. In our approach, the binary analyzer or compiler analyzes the program, perhaps with the aid of profile information, and determines the slice of the program that forms the main thread and what the dependences between the main and work threads are. The compiler then inserts explicit spawn instructions into the program as needed.

The first instruction, the unchecked spawn, *pbrnc* (non-checked parallel branch), spawns off a work thread but does not check to see whether data generated by the work thread is read back in by the main thread. The *pbrnc* instruction itself consists of the opcode and the spawn target. When a processor core (P0) encounters a *pbrnc*, it sends its architectural state to P1 (details in the next section). When P1 dequeues the spawn, it begins execution at the spawn target. P0 continues executing the instruction following the *pbrnc*. When a work thread is spawned by a *pbrnc*, it is expected that the output of the spawned task will not be accessed by the main thread. If the P0 does access a register or memory

location written by the spawned task, then it is ambiguous whether it will see the old value or the new one.

The ambiguity is alleviated by the second instruction, the checked spawn, *pbr* (checked parallel branch). Like *pbrnc*, this instruction spawns off a work thread, but it also guarantees that the main thread will see any output generated by the spawned instructions. A processing core will not retire instructions following a *pbr* until it can guarantee that those instructions see correct data (although it may execute them before the spawned task finishes).

The instruction to mark the end of execution on a work thread is the endblock, *pjn* (parallel join). When the work thread processor sees a *pjn*, it halts execution on the current path and sends back to the main thread processor a map of what architectural state it updated. At this point, the work thread processor checks to see if its spawn queue has any pending spawns, and if so it dequeues and begins executing the next spawn. If there are no pending spawns, the processor stalls.

## 3. ARCHITECTURE

In this Section we examine an example implementation of the NXA model that is suitable for fine-grained decoupling. The microarchitecture of this proposed implementation, shown in Figure 3.1 is based on a dual-core CMP. Each core is a deeply-pipelined, out-of-order superscalar processor. Added to this CMP are the spawn queues plus a data communication network. These added blocks are shown in dark gray.

### 3.1. Spawn Queuing

The NXA main spawn queue is implemented as a FIFO connecting the rename stage of P0 with the fetch state of P1. When P0 decodes a *pbr* or *pbrnc* instruction, it sends the target of the spawn to the FIFO. At the fetch stage, P1 initiates fetch from the target of the spawn. If P1 encounters a *pjn*, it initiates fetch from the next spawn in the FIFO, or stalls if no spawns are enqueued.

### 3.2. Register Communication

Register communication between P0 and P1 is performed lazily, with bitmasks used to identify which registers need to be communicated. 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. When P0 encounters a *pbr* or *pbrnc* instruction, it sends the bitmask to P1 in the spawn request along with the target PC, and resets the bitmask. Likewise, P1, when it encounters a *pjn* instruction, sends and resets its register update bitmask back to the ROB in P0. In our Alpha-based implementation, the register update mask is 76 bits wide, with a bit for each integer, floating point, and special register.

Register communication from P0 to P1 is simple. When P1 dequeues a spawn, it reads the bitmask to determine the

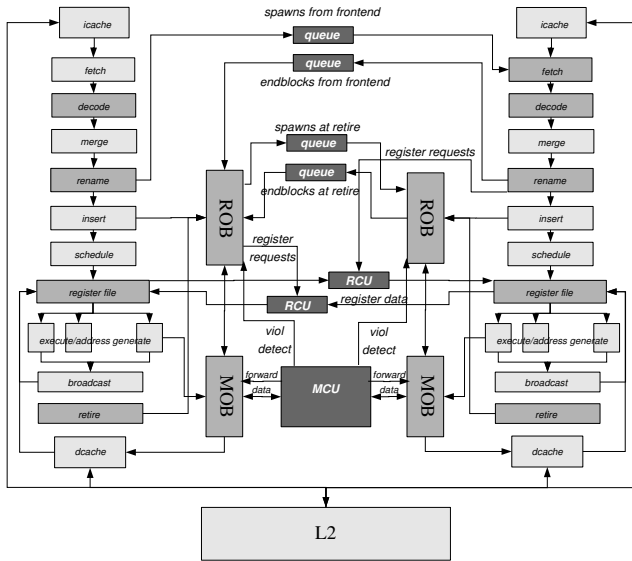


Figure 3. An NXA implementation.

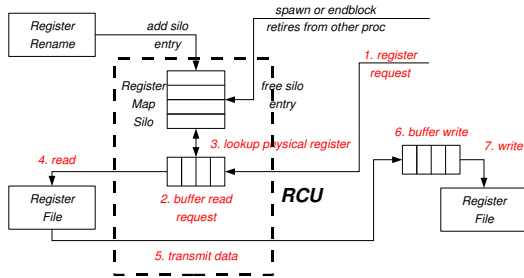


Figure 4. Register communication unit for one of the processor cores. Both cores have their own RCU.

set of registers written by P0. This set is recorded as the stale register mask. When an instruction in P1 passes the decode stage, its inputs are compared to the stale register mask to determine which must be read from P0.

Register communication from P1 back to P0 is more complicated. As stated, when P1 encounters a *pjn* instruction, it sends its register update mask to P0, along with an identifier for the originating spawn. P0 records this as its stale register mask. If the *pjn* corresponds to a checked (*pbr*) spawn, P0 checks before the retire stage to see whether instructions incorrectly read any stale registers; if so, it enqueues a request for these registers and replays any dependent instructions.

The actual register communication is carried out by the structure shown in Figure 3.2. The mechanism contains, on each processor’s side, a register map silo (containing mappings from the architectural register to the physical register at the point of each *pbr/pbrnc* or *pjn* instruction), a buffer of pending register requests originating on that processor, and a buffer of pending reads from that processor’s register file.

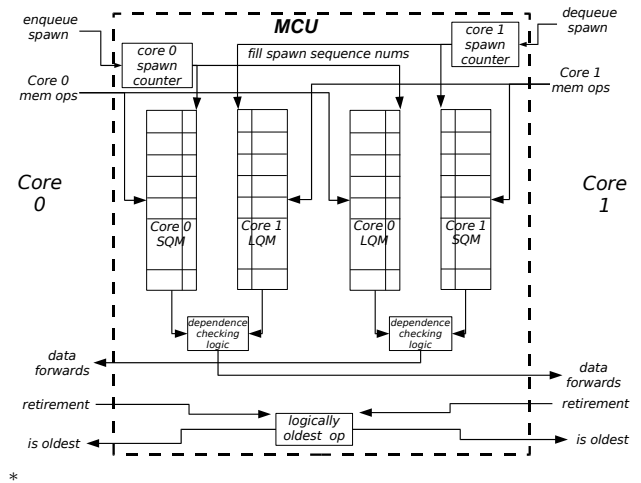


Figure 5. Memory communication unit.

### 3.3. Memory Communication

Memory communication between P0 and P1 is handled by the memory communication unit (MCU) shown in Figure 3.3. The basic idea of the MCU is to mirror the memory order buffer from each processor, and use these mirrors to detect and data dependence violations and perform inter-processor store forwarding. The structure itself consists of a load queue mirror (LQM) and a store queue mirror (SQM) for each processor. The LQM for processor core 0 is coupled to the SQM for processor core 1, and vice versa. The MCU also contains a spawn sequence counter for each core, as well as a register keeping track of the logically oldest memory operation that has not yet been committed. Finally, the MCU contains logic to detect memory dependences and perform the store forwarding.

When one processor executes a load or a store, it forwards the operation and its address to the MCU. The MCU then uses the LQM and SQM to determine whether there is a matching load or store from the opposite core. If so, it forwards the data value. The MCU also informs each core whether it contains the oldest current memory operation. In the case of a *pbr*, P0 will not commit memory operations until they are the oldest, so that the MCU can guarantee that there are no more dependence violations. This restriction does not apply with *pbrnc* instructions.

## 4. DECOUPLING IMPLEMENTATION

The NXA architecture provides a target for decoupled programming. In this Section, we examine how to implement our approach to parallelization in a compiler or static binary analyzer. Each of the decoupling schemes we present can be performed statically by data and control flow analysis on the program.

Our first method of decoupling the program is to separate the portions of the program that compute control flow from the portions that compute data values. Our second

method is to separate the memory accessing portions of the program from the consumers of those memory accesses. Our third method is to separate the dataflow critical path of the program from the non-critical portion. All three methods take advantage of the same iterative algorithm for thread selection.

#### 4.1. Parallelization Algorithm

The decomposition of the program into main and work threads is carried out in several steps. We first choose an initial set of instructions for the main thread. Then, we compute the control dependence and control flow graphs. Next, we use iterative dataflow analysis [14] to compute the dataflow and control dependence closure of the initial set, which becomes the main thread. We separate out the main thread and the work thread, and finally we insert spawn and endblock instructions to couple the threads.

##### 4.1.1. Iterative main thread selection.

Given an initial seed for the main thread, we can compute the remainder of the thread using iterative dataflow analysis. After computing the control dependence and control flow graphs, we compute the remainder of the main thread (MT) by computing the set of variables that are live in the MT at each point in the program. We define a variable as *MT\_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 MT instruction before it is overwritten. Any instruction that writes an *MT\_live* variable is part of the MT.

The computation of *MT\_live* sets is similar to live variable analysis. Like live variables, *MT\_live* variables can be computed by iterative solution of dataflow equations. The dataflow equations for *MT\_live* variables are shown below. Note that these equations are virtually identical to those for live variable analysis, except for the presence of *MT\_use*( $B$ ) in place of *use*( $B$ ).

$$MT\_live\_in(B) = MT\_use(B) \cup (MT\_live\_out(B) - kill(B))$$

$$MT\_live\_out(B) = \bigcup_{S \in Succ(B)} MT\_live\_in(S)$$

The actual computation of the *MT\_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 *MT\_live\_out*. Then, we iterate through the instructions in the block in reverse, keeping track of the *MT\_live* set. If we encounter an instruction that writes a *MT\_live* variable, we add it to the *MT\_inst\_set*. We also remove its outputs from and add its inputs to the *MT\_live* set. If we encounter an instruction already in the *MT\_inst\_set*, we similarly remove its outputs from and add its inputs to the *MT\_live* set, and in addition we add any instruction it is control dependent on to the *MT\_inst\_set*. At the top of the basic block we union *MT\_live\_in* with the *MT\_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 *MT\_inst\_set* or any of the *MT\_live* sets.

##### 4.1.2. Profile-directed optimizations.

In many cases, dataflow dependences that are present in the static code are not significant in the dynamic execution of the code. This is because many of the code paths in the program are rarely or never taken, and so dependencies across those paths never occur. We can identify which program paths are significant through profiling, and use the information to reduce the size of the main thread and hence improve its performance.

One such optimization is infrequent path pruning (IPP). Using profile information to identify infrequently taken paths, we remove infrequent edges from the CFG while computing MT liveness. If the profile information is accurate, IPP will remove unnecessary instructions from the MT more often than it introduces dependence violations.

A second profile directed optimization is biased branch exclusion (BBE). To apply BBE, we first determine from the profile which branches are highly biased, and add these branches to a table. Then, during the *MT\_liveness* analysis, we propagate the *MT\_liveness* of a variable only if its reader is not one of the biased branches. Finally, we mark each biased branch for delayed resolution.

##### 4.1.3. Spawn Coalescing.

After the MT selection has taken place, we wind up with code that often interleaves MT and WT instructions. As each spawn instruction requires dependence checking, this can be wasteful. Spawn coalescing reorders contiguous blocks of MT and WT instructions to minimize the number of transitions between the two, and hence reduce the number of spawn (and endblock) instructions.

##### 4.1.4. Checked spawns.

Profile-directed optimizations introduce backwards data dependences, where code in the main thread may depend on data values produced by the work thread. Though these dependences should only be realized across infrequently executed paths, we still need to check for violations with *pbr* checked spawns. To determine whether each spawn should be a *pbrnc* or *pbr*, we unprune the CFG and perform the *MT\_liveness* analysis again. Wherever an instruction previously identified as being in the WT writes a value that is now *MT\_live*, we must use a *pbr*.

#### 4.2. Control Decoupling

Studies such as [11] have shown that control flow is a major bottleneck limiting the available instruction-level parallelism. A strategy to alleviate this bottleneck is to accelerate the computation of control decisions. On NXA, we can perform this by putting the control flow computation in the main thread, as a control thread, and deferring data computation to the work thread. By doing so, we alleviate

resource constraints limiting the performance of the control-flow computing portion of the program.

To compute the control thread, we use the program’s control instructions (i.e. branches, jumps, subroutine calls) as the seed for the iterative thread selection.

We can improve control decoupling with Local Control Exclusion (LCE), an optimization where we move “local” control flow to the work thread while reserving “global” control flow for the control thread. Because these local control flow constructs are not part of the control flow bottleneck, LCE can reduce the complexity of the control thread without increasing the control flow bottleneck. This can benefit performance significantly when, as is usually the case, the control thread would otherwise contain a very large fraction of the total program.

We perform LCE over three specific “local” control flow constructs: if-then-else constructs (ITE), tight loops consisting of a single basic block (TL), and loops containing an if-then-else (ITEL). This is done by excluding anything within these constructs from the iterative selection seed.

### 4.3. Memory Decoupling

As noted, control flow is a major bottleneck constraining the available parallelism. Another bottleneck is memory access. When a memory access suffers a long latency miss, all its dependent instructions are delayed. Because of resource constraints in real superscalar machines, this often delays surrounding instructions as well.

There are two solutions to this problem. One solution, seen in work on memory helper threads [3] [12] [4], is to perform miss-prone loads in advance of the main program execution, to alleviate the latency of such misses. Another solution, which we propose here, is to defer instructions not depended on by miss-prone loads, to alleviate the resource constraints that can delay the main execution flow. In this case the main thread is the memory thread, and the deferred instructions are the work thread.

We perform memory decoupling on the NXA architecture with a 2 step process. The first step is to examine the program and determine the miss-prone loads. These miss-prone loads are used as the seed for computing the memory thread.

To identify miss-prone loads, we ran each benchmark through a cache simulator, which provided the frequency with which each memory operation missed in the cache and also the cumulative latency incurred by those memory operations. The memory thread seed can be selected from this information by setting a threshold for the cumulative latency (i.e. an instruction is in the initial memory thread if it incurred a cumulative latency above a certain threshold) or by miss count (i.e. an instruction is in the initial memory thread if the miss count was above a certain threshold).

Several of the optimizations presented for control decoupling work the same way for memory decoupling. In

particular, we can use IPP, BBE, and spawn coalescing unchanged from control decoupling. LCE, however, is not applicable since our initial main thread does not consist of control instructions.

### 4.4. Critical Path Decoupling

Finally, dataflow dependences themselves are a major constraint on achievable parallelism. Thus, just as we used work threads to defer non-control instructions and non-memory instructions, we can use them to defer non-critical path instructions.

The key to selecting the critical path thread is determining the critical path through the execution. There are several classes of “critical path” that can be obtained from a dynamic execution of the program. One is the longest spine of the dataflow tree height. Another is the longest spine of the dataflow tree height augmented with control dependences, in a manner similar to Lam and Wilson [11]. A third is dataflow+CD+memory latencies. Fourth, we can use dataflow+memory latencies, without control dependences. Beyond this, we can even use a critical path predictor similar to the one in [5]. In this paper we focus on dataflow+memory as experimentation showed it to provide the best performance.

Once we compute the critical path we want, we use the static instructions corresponding to that path as the seed for computing the critical path thread, using the same iterative selection algorithm as for control and memory thread selection.

As before, we can apply infrequent path pruning and BBE to critical path thread selection. In addition, we can reduce the size of the seed by noting that some static instructions occur in the dynamic critical path just a few times or even only once. In such cases, the penalty of having that instruction and its dependences in the critical path thread during all the non-critical executions might be greater than the benefit of having the instruction in the critical path thread during the few critical executions. Hence, we can eliminate from the seed instructions that are critical only once (the crit2 optimization).

## 5. PERFORMANCE EVALUATION

In this section, we use microarchitectural simulation to evaluate the performance of NXA with the three decoupling techniques.

### 5.1. Methodology

We simulate the performance of NXA using Joshua, a detailed microarchitectural simulator we have configured to execute Alpha binaries. Joshua simulates an array of aggressive out-of-order superscalar processors in a chip multiprocessor. Here, Joshua has been configured to simulate the NXA implementation described in Section 3. The parameters for the simulated baseline machine are given in Table 1. Additional parameters for the NXA machine are given in Table 5.1.

**Table 1. Simulated 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
L0 Cache	8KB/1 cycle latency instruction and 16KB/1 cycle latency data
L1 Cache	128KB/8 cycle latency unified
L2 Cache	4MB, 20 cycle latency
Memory	200 cycle latency
Branch prediction	2K entry BTB plus 1Mbit hybrid predictor

**Table 2. Simulated NXA parameters**

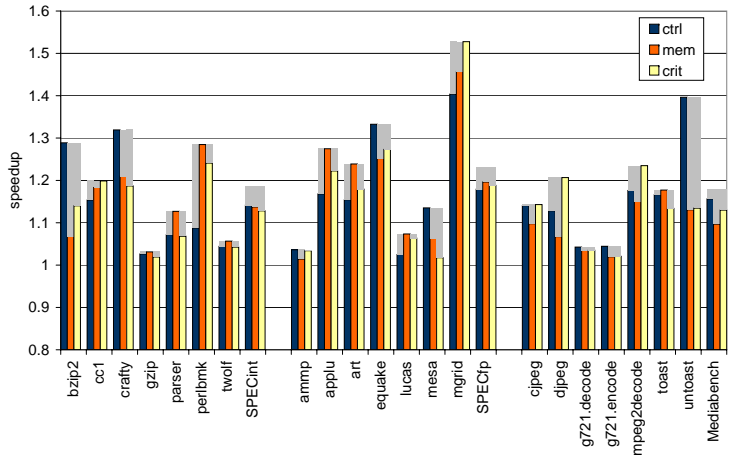
NXA parameter	value
Spawn queue size	256 spawns
Register communication queue	128 pending reads per processor
Register communication latency	2 cycles minimum
Register communication bandwidth	2 communications per cycle
Memory communication unit	320 total entries
Memory bypass latency	5 cycles minimum
Memory bypass bandwidth	1 bypass per cycle
Cache coherence	update protocol

As workloads, we used 7 benchmarks each from SPECint2000, SPECfp2000, and MediaBench. We used reduced MinneSPEC [9] input sets for the SPEC benchmarks, and the standard input data set for MediaBench. We use binaries compiled with the Compaq Alpha C V5.9 and C++ V6.5 compilers with optimization level -O4. All SPEC benchmarks were run for 200M instructions, while MediaBench benchmarks were run through to completion. Profile information was gathered from the first half of the performance run.

We perform the control, memory, and critical path decoupling via static binary analysis. The binary analyzer takes as input the program binary, decoded into an internal format, (optionally) the branch bias information from profiling, and finally the main thread seed generated by the memory or critical path analysis.

## 5.2. Performance results

Figure 5.2 shows the performance results by benchmark. The three foreground bars show, for each benchmark, the speedup obtained by control, memory, and critical path decoupling. Table 5.4 shows the combination of optimizations that provided this speedup for each benchmark. The gray background bars show the speedup attained by the best of the three decoupling models. The results show an average speedup of 1.16 for control decoupling, 1.14 for memory decoupling, and 1.15 for critical path decoupling. When



**Figure 6. Performance of best decoupling scheme for each benchmark.**

the best decoupling method (shown in the last column of Table 5.4) is applied to each benchmark, the average speedup becomes 1.20.

## 5.3. Effect of decoupling schemes

From Figure 5.2, we observe that different benchmarks perform best with different decoupling schemes. In general, we expect benchmarks that are control-flow bound to perform best with control decoupling, benchmarks that are memory bound to perform best with memory decoupling, and benchmarks that are bound by dependent chains of instructions to perform best with critical path decoupling.

By “control-flow bound”, we mean those programs where being able to accelerate the computation of control flow decisions exposes significant additional parallelism. Performing well with control decoupling are bzip2, crafty, equake, and mesa, as well as untoast from Mediabench. Bzip2 and equake consist of tight loops with a relatively small amount of code. The control flow within these loops can be computed in parallel with the data generation along at least the more frequent paths. Most of mesa runs in nested loops where the inner loop is a tight loop that can be spawned off to the work thread. With untoast, control decoupling with LCE just manages to distribute the workload very evenly across the two processing cores, and control dependences are less of a factor.

The benchmarks that perform best with memory decoupling are gzip, parser, perlbmk, twolf, applu, art, lucas, and toast. Of these gzip, twolf, and lucas perform weakly on all the decoupling schemes. However, in parser, perlbmk, and art, upwards 30% of instructions are memory operations, and they are indeed memory bound. Applu has fewer memory operations, but it has a low hit rate in the dcache of just 86%. Toast on the other hand is not memory bound; in fact, toast is ALU bond and memory decoupling simply did a better job of distributing the code from toast across the

ALUs of the two processing cores than the other decoupling schemes.

The benchmarks performing best with critical path decoupling are gcc (cc1), mgrid, and jpeg/mpeg. Gcc has a large number of dependence chains that flow through memory operations that miss in the caches; when we compute the critical path using the dataflow+memory model, these chains appear in the critical path. Hence, critical path decoupling helps gcc performance by moving chains of dependences through miss-prone loads to the main thread and deferring chains through non-miss-prone loads to the work thread, reducing pressure on the load/store system during the critical sections of the code. On the other hand, mgrid is bound by the number of FP ALUs, and (similar to toast) critical path decoupling just does the best jobs of distributing the code across the ALUs. The reason is similar for the jpeg/mpeg benchmarks.

#### 5.4. Effect of optimizations

Figure 5.4 shows the speedups obtained by specific decoupling methods with specific optimizations applied to all the benchmarks. From the graph, several trends can be observed.

Infrequent path pruning tends to benefit SPECint a lot, have mixed results on MediaBench, and hurt performance on SPECfp. The integer benchmarks tend to have a lot of control instructions and hence a lot of control dependence edges. Pruning helps cut down on the number of instructions that get included in the main thread due to those dependences. In the FP benchmarks, there are relatively few control instructions; hence, decreasing the size of the main thread by pruning hurts load balance. In MediaBench, some benchmarks observe the integer trend, while others observe the FP trend.

BBE has a mixed impact. On many benchmarks (especially FP), the reverse dependences introduced by the optimization impede performance much more than the reduction in the main thread’s complexity. However, on some integer benchmarks such as crafty and gcc, there are highly biased, easily predicted branches whose inputs have a long chain of dependences. In this case, there is a substantial gain from being able to move this whole chain over to the work thread.

Overall, the results of Figure 5.4 illustrate the potential usefulness of heuristics to determine automatically which optimizations to apply.

#### 5.5. Sensitivity studies

To further evaluate the NXA architecture design space, several hardware parameters were varied to determine their impact on performance. Two classes of parameters were investigated: inter-core communication latency and bandwidth. The best performing partitioning scheme for each benchmark was used in this study.

None of the communication parameter changes had an appreciable effect (<1%) on performance. Register com-

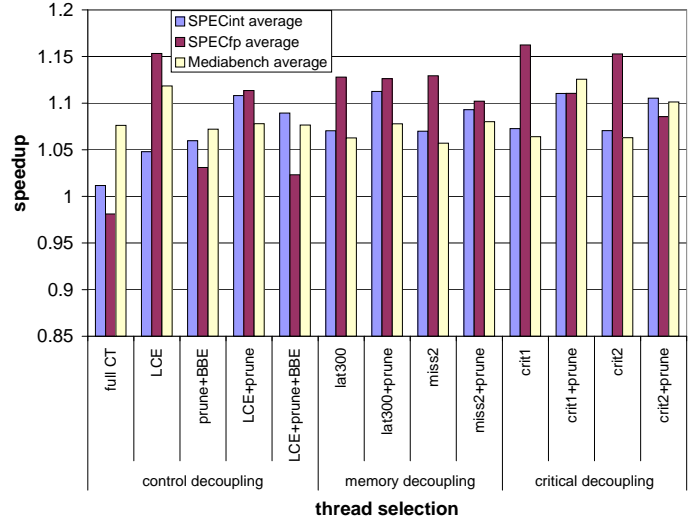


Figure 7. Speedup with different decoupling methods and optimizations.

munication latency was increased up to 8 cycles while bandwidth was varied between 1 and 4 communications per cycle. Memory communication latency was increased up to 15 cycles while (independently) bandwidth was upped to 4 communications per cycle. The result gives evidence that our partitioning schemes work well to keep inter-thread communication flowing in only one direction.

## 6. RELATED WORK

This work extends a large body of work on decoupled architectures. Previous work in this area can be roughly classified into three categories: the early work based on pre-modern processor (non-superscalar or short pipeline) architectures; speculative multithreading, which attempts to find relatively coarse-grained, thread-level parallelism; and the slicing approach, where threads consist of “slices” of dependent instructions. Our hardware design borrows features from speculative multithreading, but differs greatly in the granularity of parallelism that we can exploit. Moreover, our work uses the slicing technique for generating the threads, but executes those threads in a fundamentally different manner, hence constituting a fourth category.

### 6.1. Early work

One of the first proposals for decoupled architectures was the Decoupled Access/Execute Architecture [17], where the memory address stream was decoupled from the execution stream. This enabled memory references to slip ahead of execution and provide a prefetching effect. The PIPE [7] work provides a more thorough evaluation of DAE. The ACRI work [1] was another early proposal for decoupled architecture. Bird et al. introduced the term “control decoupling”, proposing to not only decouple the memory address

**Table 3. Best combination of optimizations for each benchmark.**

benchmark	ctrl	mem	crit	overall
bzip2	all LCE+prune	lat300+prune	crit2+prune	ctrl
cc1	all opts	lat300+prune	crit2+prune	crit
crafty	all opts	lat300+prune	crit2+prune	ctrl
gzip	ITEE+TLE	lat300+prune	crit2+prune	mem
parser	all LCE+prune	lat300+prune	crit2+prune	mem
perlbmk	all LCE	lat300	crit2	mem
twolf	all opts	miss2+prune	crit2+prune	mem
ampp	prune+BBE	miss2+prune	crit2+prune	ctrl
applu	all LCE	miss2	crit1	mem
art	ITEE	lat300+prune	crit1	mem
equake	all LCE+prune	lat300+prune	crit1+prune	ctrl
lucas	all LCE	miss2	crit2	mem
mesa	ITEE+TLE	lat300+prune	crit1	ctrl
mgrid	all LCE	miss2	crit1	crit
cjpeg	all opts	lat300+prune	crit2+prune	crit
djpeg	all LCE+prune	miss2+prune	crit1+prune	crit
g721.dec	all opts	miss2+prune	crit2+prune	ctrl
g721.enc	all LCE	miss2+prune	crit1+prune	ctrl
mpeg2dec	ITEE	lat300	crit1+prune	crit
toast	all opts	lat300+prune	crit2+prune	mem
untoast	ITEE+TLE	miss2+prune	crit1+prune	ctrl

stream, but also the control stream. Our work generalizes these techniques by evaluating several different methods of program partitioning rather than focusing specifically on the memory access or control stream.

## 6.2. Speculative multithreading

More recently, a variety of speculative multithreading architectures have been proposed. Speculative multithreading attempts to get around the limitations of traditional parallelizing compilers, which can only parallelize codes when they can guarantee correctness. The idea of speculative multithreading is to go ahead and parallelize even when the compiler is not sure, and then use hardware to squash the parallelism if it turns out incorrect.

An early example of a speculative multithreaded architecture is Multiscalar [6] [18]. Whereas Multiscalar required static analysis, SM processors proposed in [13] attempted to achieve parallelism using only control speculation in hardware. This technique achieved strong performance on FP codes but fell flat in parallelizing integer codes. More recent proposals, including [16], [10] and [19], propose mechanisms that require less hardware overhead. In [16], Olukotun et al. describe how to support fine-grain speculative parallelism on the Hydra CMP. In [10], a CMP design is proposed that adds a modest set of structures for speculative multithreading. 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 [19] and [20], Steffan et al. describe an approach that imposes even less hardware overhead, by ex-

tending cache coherence mechanisms to detect dependence violations.

The main difference between NXA and speculative multithreading is the method of thread selection and the source of the performance improvement. Instead of partitioning the dynamic instruction stream into relatively large contiguous regions based on control independence, we finely slice the instruction stream, isolating critical portions of control and data flow. By separating the worker threads from the main thread, we prevent non-critical work from competing for resources and slowing overall execution. Because we deal with finer grained parallelism, we are also able to reduce the degree of speculation by eliminating speculative spawns and hence simplify the hardware.

## 6.3. Slicing approaches

Still more recently, research in decoupled architecture has focused on slicing approaches. These approaches are exemplified by the various work on prefetching helper threads [3] [12] [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 and reduces the cache miss rate of the main thread. The branch-decoupled architecture [22] [15] attempts a similar feat on branches, where the branch outcomes from the branch thread serve as branch predictions for the main thread.

The Slipstream processor [21] [8] is another example of the slicing approach. In Slipstream, one processor executes a filtered program called the *advanced stream* (A-stream), using it to generate intermediate results that are fed to another processor running the original program (called the R-stream).

A final example of the slicing approach is master-slave speculative parallelization [23] [24]. In MSSP, there is one main thread that operates on speculative data and one or more lagging threads that run in parallel and verify the data. The main thread thus focuses on data computation while the slave threads are responsible for verifying much of the control flow.

Each of these slicing approaches makes the observation that either control flow, memory accesses, or error checking code can be pulled out of the program and run in parallel with it. With NXA we exploit all three observations with variations on a common parallelization algorithm and a single architecture.

At an implementation level, all of the previous slicing approaches share the same basic approach. Some slice/subset of the program is taken out and executed as a speculative thread. Then, another thread or multiple threads comes along and executes the entire program non-speculatively, using the speculative slice to speed its execution. Hence, all the approaches use a speculative thread/verification thread(s) paradigm. NXA does not. With NXA the chosen slice is not a prediction thread but rather the

main thread of execution. The NXA approach has significant advantages. NXA manages to exploit the same underlying parallelism, but it does so without redundant instructions or speculative threads, reducing resource constraints during execution and simplifying the hardware implementation.

## 7. Conclusion

Automatic parallelization of sequential code is difficult on conventional chip multiprocessors. In this paper, we offer three contributions to alleviate the limitations:

1. NXA<sup>†</sup>, a novel multi-core architecture based on fine-grained parallelism within a single thread of control. Previous approaches lacked the ability to exploit fine grained threads.

2. A unified parallelism model that generalizes slice-based approaches. We use a single architecture to exploit observations made by separate prior work that control flow, memory accesses, or error checking code could be pulled out of the program and executed in parallel with it.

3. Effective algorithms for parallelizing a non-parallel application using a fine-grained architecture.

We evaluate the performance of control, memory, and critical path decoupling on the NXA architecture through microarchitectural simulation and demonstrate that the combination does indeed offer performance benefits. By choosing threads with the best combination of optimizations for each benchmark, we obtained speedups of 1.19 on SPECint2000, 1.23 on SPECfp2000, and 1.18 on MediaBench when compared against an aggressive 4-wide super-scalar processor.

## References

- [1] P. L. Bird, A. Rawsthorne, and N. P. Topham. The effectiveness of decoupling. In *Proceedings of the 7th International Conference on Supercomputing*, pages 47–56, 1993.
- [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. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, 2001.
- [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] B. A. Fields, S. Rubin, and R. Bodik. Focusing processor policies via Critical-Path prediction. pages 74–85.
- [6] M. Franklin. The multiscalar architecture. Technical Report 1196, Computer Sciences Department, University of Wisconsin - Madison, Nov. 1993.
- [7] J. R. Goodman, J. tu Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. Pipe: A vlsi decoupled architecture. In *ISCA*, pages 20–27, 1985.
- [8] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 179–190, 2003.
- [9] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research, 2002.
- [10] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, Sept. 1999.
- [11] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 248–259, 1992.
- [12] C. Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, 2001.
- [13] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, pages 77–84, 1998.
- [14] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [15] A. S. Nadkarni and A. Tyagi. A trace based evaluation of speculative branch decoupling. In *IEEE International Conference on Computer Design ICCD'2000*, pages 300–, 2000.
- [16] 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.
- [17] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.
- [18] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.
- [19] 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.
- [20] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, pages 65–, 2002.
- [21] 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.
- [22] A. Tyagi, H.-C. Ng, and P. Mohapatra. Dynamic branch decoupled architecture. In *IEEE International Conference on Computer Design ICCD'1999*, pages 442–450, 1999.
- [23] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, 2001.
- [24] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002.

<sup>†</sup>New eXciting Architecture