Exploiting Fine–Grain Thread Level Parallelism on the MIT Multi-ALU Processor

Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, Whay S. Lee†

Computer Systems Laboratory
Stanford University
Gates CS Building
Stanford, CA 94305

†Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

{skeckler, billd, danielm, achang, npcarter, wslee}@cva.stanford.edu

Abstract

Much of the improvement in computer performance over the last twenty years has come from faster transistors and architectural advances that increase parallelism. Historically, parallelism has been exploited either at the instruction level with a grain-size of a single instruction or by partitioning applications into coarse threads with grain-sizes of thousands of instructions. Fine–grain threads fill the parallelism gap between these extremes by enabling tasks with run lengths as small as 20 cycles. As this fine–grain parallelism is orthogonal to ILP and coarse threads, it complements both methods and provides an opportunity for greater speedup. This paper describes the efficient communication and synchronization mechanisms implemented in the Multi-ALU Processor (MAP) chip, including a thread creation instruction, register communication, and a hardware barrier. These register-based mechanisms provide 10 times faster communication and 60 times faster synchronization than mechanisms that operate via a shared on-chip cache. With a three-processor implementation of the MAP, fine–grain speedups of 1.2–2.1 are demonstrated on a suite of applications.

1 Introduction

Modern computer systems extract parallelism from problems at two extremes of granularity: instruction-level parallelism (ILP) and coarse-thread parallelism. VLIW and superscalar processors exploit ILP with a grain size of a single instruction, while multiprocessors extract parallelism from coarse threads with a granularity of many thousands of instructions.

The parallelism available at these two extremes is limited. The ILP in applications is restricted by control flow and data dependencies [17], and the hardware in superscalar designs is not scalable. Both the instruction scheduling logic and the register file of a superscalar grow quadratically as the number of execution units is increased. For multicomputers, there is limited coarse thread parallelism at small problem sizes and in many applications.

* The research described in this paper was supported by the Defense Advanced Research Projects Agency and monitored by the Air Force Electronic Systems Division under contract F19628-92-C-0045.

This paper describes and evaluates the hardware mechanisms implemented in the MIT Multi-ALU Processor (MAP chip) for extracting fine–thread parallelism. Fine–threads close the *parallelism gap* between the single instruction granularity of ILP and the thousand instruction granularity of coarse threads by extracting parallelism with a granularity of 50-1000 instructions. This parallelism is orthogonal and complementary to coarse-thread parallelism and ILP. Programs can be accelerated using coarse threads to extract parallelism from outer loops and large co-routines, fine–threads to extract parallelism from inner loops and small sub–computations, and ILP to extract parallelism from subexpressions. As they extract parallelism from different portions of a program, coarse–threads, fine–threads, and ILP work synergistically to provide multiplicative speedup.

These three modes are also well matched to the architecture of modern multiprocessors. ILP is well suited to extracting parallelism across the execution units of a single processor. Fine–threads are appropriate for execution across multiple processors at a single node of a parallel computer where the interaction latencies are on the order of a few cycles. Coarse-threads are appropriate for execution on different nodes of a multiprocessor where interaction latencies are inherently 100s of cycles.

Low overhead mechanisms for communication and synchronization are required to exploit fine–grain thread level parallelism. The cost to initiate a task, pass it arguments, synchronize with its completion, and return results must be small compared to the work accomplished by the task. Such inter-thread interaction requires 100s of cycles ($\geq 1\mu$s) on conventional multiprocessors, and 1000s of cycles ($\geq 10\mu$s) on multicomputers. Because of these high overheads, most parallel applications use only coarse threads, with many thousands of instructions between interactions.

The Multi-ALU Processor (MAP) chip provides three on-chip processors and methods for quickly communicating and synchronizing among them. A thread executing on one processor can directly write to a register on another processor. Threads synchronize by blocking on a register that is the target of a remote write or by executing a fast barrier instruction.

Microbenchmark studies show that with these register-based mechanisms, a thread can be created in 11 cycles, and individual communication and synchronization actions can be performed in 1 cycle. Communication is 10 times faster and synchronization is 60 times faster than their corresponding memory mechanisms

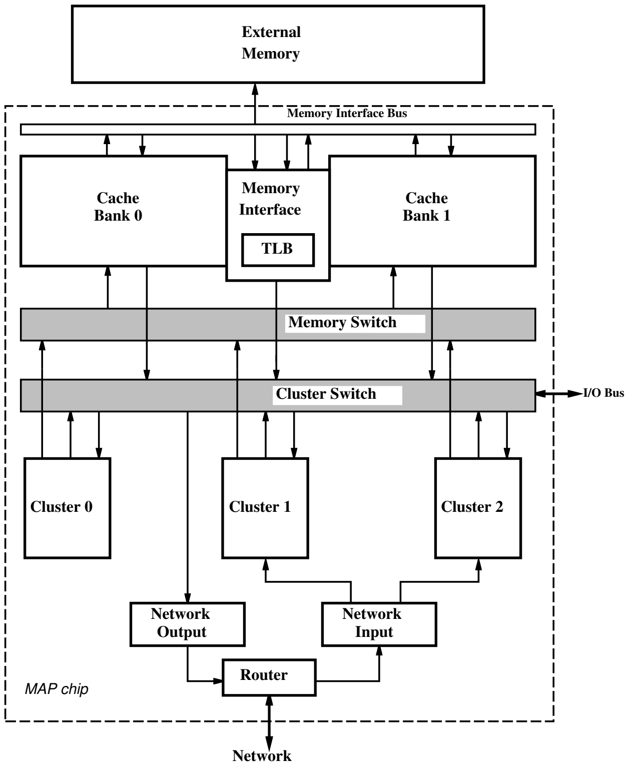

Figure 1: Block diagram of the MAP chip, containing 3 clusters (processors) connected to the memory system and each other via the Memory and Cluster switches.

that use the on-chip cache. The MAP's integrated mechanisms are orders of magnitude faster than those that operate via global memory. A study using several parallel applications shows that these mechanisms allow speedups of up to 2.1 using 3 on-chip processors when exploiting parallelism with a granularity of 80–200 cycles. If the register communication provided by the MAP is replaced with memory operations, fine–grain threads yield substantially less speedup, and sometimes slowdown.

The next section describes the architecture and implementation of the MAP chip and details its mechanisms that support fine–grain threads. The raw performance of these mechanisms is evaluated in Section 3 and compared to memory-based mechanisms. Section 4 explores the granularity of several parallel applications and the speedup realized by exploiting fine–grain threads with and without the MAP chip mechanisms. Related research that addresses fine–grained and on–chip parallelism is described in Section 5, and concluding remarks are found in Section 6.

2 The MAP Chip

The Multi-ALU Processor (MAP) chip, designed for use in the M-Machine Multicomputer, is intended to exploit parallelism at a spectrum of grain sizes, from instruction level parallelism to coarser grained multi-node parallelism [5]. It employs a set of fast communication and synchronization mechanisms that enable execution of fine–grain parallel programs.

Figure 1 shows a block diagram of the MAP chip containing three execution clusters, a unified cache which is divided into two banks, an external memory interface, and a communication subsystem consisting of network interfaces and a router. Two crossbar switches interconnect these components. Clusters make memory requests to the appropriate bank of the interleaved cache over the 142-bit wide (51 address bits, 66 data bits, 25 control bits) 3×2 Memory Switch. The 88-bit wide (66 data bits, 22 control bits) 7×3 Cluster Switch is used for inter-cluster communication and to return data from the memory system. Each cluster may transmit on the Memory Switch and receive on the Cluster Switch one request per cycle. An on-chip network interface and two-dimensional router allow a message to be transmitted from a cluster's register file into the network. Multiple MAP chips can be connected directly together in a two-dimensional mesh to construct an M-Machine multicomputer. This paper focuses on the parallelism that can be exploited within a single MAP chip. Future studies will analyze application performance on the M-Machine.

MAP Execution Clusters: Each of the three MAP clusters is a 64-bit, three-issue, pipelined processor consisting of two integer ALUs, a floating–point ALU, associated register files, and a 4KB instruction cache[1]. Each integer register file has 14 registers per thread, while the floating–point register file has 15 registers per thread. Each thread also has 16 condition code (CC) registers to hold boolean values. Writes to a subset of the condition code registers are broadcast to the remote clusters. One of the integer ALUs in each cluster serves as the interface to the memory system. Each MAP instruction contains 1, 2, or 3 operations. All operations in a single instruction issue together but may complete out of order. No hardware branch prediction is performed on the MAP, and branches have three delay slots, due to the three pipeline stages before execution. However, since all operations may be conditionally executed based on the one-bit value of a condition code register, many branches can be eliminated. In addition, branch delay slots can be filled with operations from both sides of the branch as well as with operations from above the branch.

The execution units are 5-way multithreaded with the register files and pipeline registers of the top stages of the pipeline replicated. A synchronization pipeline stage holds instructions from each thread until they are ready to issue and decides on a cycle-by-cycle basis which thread will use the execution unit. While this enables fast, zero-overhead interleaving on each cluster, the remainder of this paper uses only one thread per cluster.

Memory System: As illustrated in Figure 1, the 32KB unified on–chip cache is organized as two 16KB banks that are word-interleaved to permit accesses to consecutive addresses to proceed in parallel. The cache is virtually addressed and tagged. The cache banks are pipelined with a three-cycle read latency, including switch traversal. Each cluster has its own 4KB instruction cache which fetches instructions from the unified cache when instruction cache misses occur.

The external memory interface consists of the synchronous DRAM (SDRAM) controller and a 64 entry local translation looka-

[1] In the silicon implementation of the MAP architecture, only cluster 0 has a floating–point unit, due to chip area constraints. The simulation studies performed in this paper include floating–point units for each of the three clusters.

Operation	Latency (cycles)
Cache hit	3
Cache miss	8–15
Branch penalty	3
FP Multiply	4
FP Add	2
FP Divide	20

Table 1: MAP chip latencies.

side buffer (LTLB) used to cache local page table entries. Pages are 512 words (64 8-word cache blocks) in size. The SDRAM controller exploits the pipeline and page modes of the external SDRAM and performs single error correction and double error detection on the data transferred from external memory. Each MAP word in memory is composed of a 64-bit data value, one synchronization bit, and one pointer bit. A set of special load and store operations specify a precondition and a postcondition on the synchronization bit and are used to construct fine–grain synchronization and atomic read–modify–write memory operations.

Table 1 shows the nominal hardware latencies of the MAP execution units and cached memory system. The cache latencies assume no switch conflicts to or from the memory system. The miss latency can vary depending on which of the pipeline or page modes can be used to access the SDRAM.

2.1 Thread Control

Invoking a thread on a remote processor is typically an expensive operation, requiring thousands of instructions to set up a stack and initialize system data structures. Remote procedure call times are typically in the microsecond range. The MAP chip implements a fast `hfork` instruction which invokes a thread on a remote cluster by automatically writing a remote program counter and updating the thread control registers. The example below starts a thread on cluster 1, using the address contained in local integer register `i8` as the instruction pointer. If cluster 1 is already executing, then `false`, indicating failure of the instruction, is returned via the Cluster Switch to condition code register `cc0`.

```
hfork i8, #1, cc0;
```

Combined with the ability to write directly to the registers of the remote cluster, the `hfork` instruction allows a remote procedure to be started in 11 cycles, which includes time to fetch the code and prime the pipeline at the remote cluster. If still faster invocation is required, a standby handler thread can be started in a remote cluster, which waits to be signalled to execute. When an instruction pointer is written to a specific register in the remote cluster, the handler jumps to the code and begins executing.

2.2 Communication

In coarse grained multiprocessors, communication between threads is exposed to the application as memory references or messages, both of which require many cycles to transmit data from one chip to another. In the MAP chip, threads on separate clusters may communicate either through the shared memory, or through registers. Since the data need not leave the chip to be transferred from one thread to another, communication is fast and well suited to fine–grain threads.

The on-chip cache allows large quantities of common storage to be readily available to each thread. Any of these locations can be used to communicate data, and the producer and consumer need not be running in near synchrony. In the MAP, a value can be communicated in memory via a load and store. The latency between the producer and consumer of the data can be as little as 10 cycles, if both accesses hit in the cache, or as long as 36 cycles, if both miss. Additional overhead is required to construct or retrieve the addresses used to communicate.

The MAP chip also implements register–register communication between clusters, allowing one cluster to write directly into the register file of another cluster, via the Cluster Switch. Register–register transfers are extremely fast, requiring only one more cycle to write to a remote register than to a local register. The result of any arithmetic operation may be sent directly to a remote register, without interfering with memory references or polluting the cache. Since the size of the register file limits the storage for communicated values, register communication is particularly suited to passing small amounts of data quickly, such as transferring signals, arguments, and return values between threads. One drawback is that register communication requires an additional synchronization between the consumer and the producer to prevent values in the destination cluster from being illegally overwritten.

2.3 Synchronization

In a concurrent system, synchronization must be used to indicate when a task is to be started, when it is complete, or when two running threads must communicate. The MAP chip allows synchronization through memory, registers, and a hardware barrier instruction, each of which has different costs and benefits.

Memory Synchronization: In the MAP chip, every memory location has a single synchronization bit that exists both in the off-chip DRAM and in the cache, enabling locking on a location-by-location basis. Special load and store operations allow atomic testing and setting of the bit.

The code fragment below shows how a spin-lock may be implemented using the memory synchronization bits. The load and synchronize operation (`ldsu`) loads the value at the address held in register `i8`, into `i9`. In the memory system the synchronization bit is compared to the precondition `pre_1`. If they are the same, the operation succeeds: the synchronization bit is set to `post_0`, the contents of the location are returned to `i9`, and the value `true` is returned to condition code register `cc0`. Otherwise, the memory contents remain unchanged, and `false` is returned to the condition code register. The subsequent branch will cause the loop to spin until the operation succeeds.

```
_loop:
   ldsu pre_1, post_0, i8, i9, cc0;
   cf cc0 br _loop:
```

A similar sequence is used to store a value and set the synchronization bit. This synchronization mechanism can be incorporated

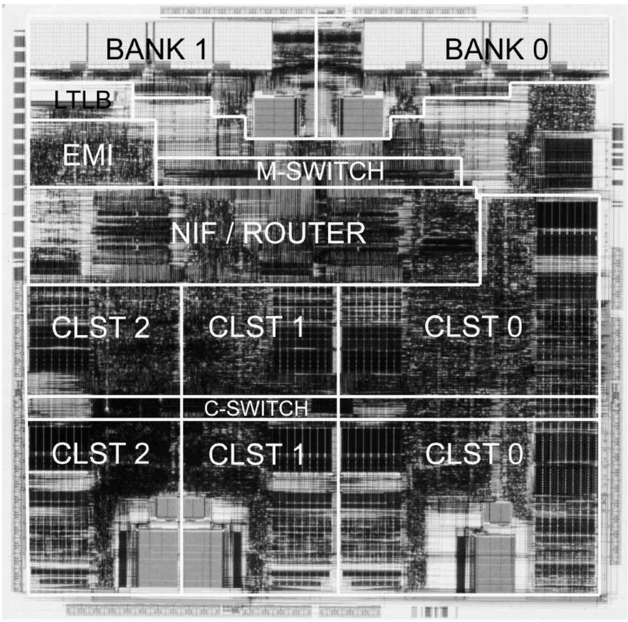

Figure 2: Preliminary plot of the MAP chip, measuring $18mm$ on a side and containing approximately 5 million transistors.

Component	Area (mm^2)	% of total area
Integer Units (3)	55.9	16.7
Memory Units (3)	42.4	12.7
16KB Data Cache Banks (2)	36.9	11.0
Floating–point Unit	33.4	10.0
NIF/Router	26.8	8.1
I/O Pads	26.6	8.0
Instruction Caches (3)	17.7	5.3
EMI + 64 entry TLB	8.3	2.5
Clock drivers	5.7	1.7
Switch drivers	3.1	0.9
Misc. Control/Wiring		23.1

Table 2: Area costs for the components of the MAP chip.

with memory communication between threads, allowing synchronization on a word by word basis. However, a consumer thread waiting for a producer will continue to make memory requests while spinning, which can slow down other threads trying to access the memory system. An alternative to spinning that can be implemented in the MAP is to invoke a software handler to retry the memory access when a synchronization failure is detected in the memory system.

Register Scoreboard: The MAP chip uses full/empty bits in a register scoreboard to determine when values in registers are valid. When an operation issues, it marks the scoreboard for its destination register invalid, and when the result is written, the destination register's scoreboard is marked valid. Any operation that attempts to use the register while it is empty will stall until the register is valid. To reduce the amount of interaction between physically distant clusters, an operation that writes to a remote cluster does not mark its destination register invalid. Instead, the consumer must execute an explicit `empty` instruction to invalidate the destination register prior to receiving any data. When the data arrives from a remote register write, the scoreboard is marked valid and any operation waiting on the register is allowed to issue. However, the producer must not write the data before the `empty` occurs. The `empty` can be guaranteed to execute first by placing it prior to an unrelated write from the consumer to the producer, or by placing a barrier between the empty and the transfer from the producer. Using register–register communication fuses synchronization with data transfer in a single operation and allows the consumer to stall rather than spin.

Barrier instruction: The simplest synchronization mechanism implemented by the MAP is the cluster barrier instruction `cbar`. The cluster barrier instruction stalls a thread's execution until the threads on the other two clusters have reached a `cbar` instruction. Threads waiting for cluster barriers do not spin or consume any execution resources. The `cbar` instruction is implemented using six global wires per thread to indicate whether a `cbar` has been reached, and whether it has been issued. Six wires per thread are necessary in order to guarantee that successive barriers stay in synchrony across all three clusters.

2.4 MAP Implementation

A preliminary layout plot of the entire MAP chip is shown in Figure 2. The chip is 18mm square and consists of approximately 5 million transistors in a 5 metal layer, 0.7μm drawn (0.5μm effective) process. The cluster datapath and control modules occupy the bottom 60% of the chip, the network interface (NIF) and router are in the middle 15%, and the memory system is in the top 25%. The Cluster Switch runs horizontally in metal-4 at the midpoint of the clusters and consumes only 8% of the metal-3 and metal-4 routing in the cluster region. However, the wiring congestion near the Cluster Switch is significant since the switch runs over the cluster pipeline control modules. The Memory Switch is below the cache banks in the memory system region and occupies about 6% of the metal-3 and metal-4 routing resources there. Table 2 summarizes the area costs for the different components of the chip. The target clock rate for the MAP chip is 100MHz. All of the datapath circuits meet that clock rate, but static timing analysis shows a clock rate of 40MHz for the control logic. The clock rate was reduced in order to avoid performing logic optimizations for speed. The final routing of the chip is currently being completed and tapeout is scheduled for April 1998.

3 Microbenchmarks

Specific microbenchmarks are used to directly evaluate the fine–grain thread control, communication, and synchronization mechanisms of the MAP chip. The microbenchmarks are written in MAP assembly code and run on both MSIM and the MAP chip register transfer level (RTL) simulator. MSIM is a functional simulator of the MAP implemented in C, and executes 400–1000 MAP cycles per second, depending on how many clusters are active. The RTL

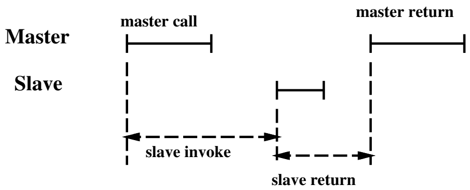

Figure 3: Components of thread invocation and return.

is the logic design of the MAP chip, implemented in Verilog. It is used for all verification, is exactly cycle accurate to the silicon, and runs less than 10 MAP cycles per second. MSIM was used to verify the logic design in the RTL and is within 5% of the cycle accuracy of the RTL over the 663 verification programs in the MAP regression suite.

3.1 Thread Creation

Three methods for starting a thread on a remote cluster, including one cold start and two standby, are examined. The cold start method uses the `hfork` instruction, which starts the remote thread automatically with a single instruction executed by the master. The standby methods already have a slave thread running in the remote cluster waiting for a new task from the master. The two standby methods differ in how the master and slave communicate with one another, using either registers or memory.

Figure 3 shows the four components of a null thread call and return. The *master call* overhead is the number of cycles that the master must spend executing instructions to create the new thread. The *slave invoke* latency is the time from the beginning of the master call to the execution of the slave's first instruction. The *slave return* latency is the time for the slave to signal to the master. Finally the *master return* is the overhead for the master to resynchronize with the slave.

Table 3 summarizes the components of latency for each of the three methods. The `hfork` instruction and the *standby register* method are the most efficient, with only 1 cycle of overhead for the master at the call and return. *Standby register* is a little faster overall as the slave invocation time is shorter. *Standby memory* is more than three times worse than the register version because of the memory spin loops the master and slave use to synchronize.

3.2 Communication

Communication latency and overhead are evaluated by using a producer-consumer microbenchmark. Both memory and register mechanisms are examined by passing a value back and forth between two clusters. The memory version uses two memory locations, one for each communication direction. Spin locks using the MAP's memory synchronization bits implement synchronization between the threads. The producer stores its value to the target location, and marks the memory location full, while the receiver spins on the location, waiting for the data to arrive. The register version uses the `empty` instruction and remote register writes. The producer empties its receiving register and writes the value to the consumer's register file. The consumer stalls on the register until the value is written and the scoreboard is marked full.

Operation	Master Call	Slave Invoke	Slave Return	Master Return	Total
hfork	1	11	2	1	14
memory	3	21	6	9	36
register	1	7	2	1	10

Table 3: Latencies for thread invocation. The total time is end–to–end latency of a null remote invocation. Using the `hfork` instruction or register communication yields an overhead three times smaller than using memory operations.

Operation	Producer Overhead	Transfer Latency
Memory (cache hit)	3	10
Register	1	2

Table 4: Communication latencies between threads.

There are two components to the efficiency of cross cluster communication. The *producer overhead* is the number of cycles that the producer must spend initiating the transfer. The *transfer latency* is the total time from the producer initiation to the use by the consumer, and includes the producer overhead. Table 4 shows the producer overhead and transfer latency for memory and register communication. Before transferring the data, the memory version must first compute the address of the communication location, which takes 3 cycles. In the register version, the remote location for the data is encoded in the instruction performing the transfer, resulting in only a single cycle producer overhead. The memory version also has a 10 cycle transfer latency, including the producer overhead and two memory latencies, one each by the producer and consumer. If either memory access misses in the cache, the transfer latency will be significantly longer. Register communication has only one additional cycle of latency for the Cluster Switch traversal, and the consumer is able to use the data immediately.

3.3 Barrier

Not all synchronization can be easily expressed using a producer-consumer model. A barrier can be used to conglomerate several synchronizations into a single action. Fast barriers reduce the overhead of using parallelism, which is vital if the parallelism to be extracted has short task execution times between synchronizations. Four implementations of barriers across three clusters are examined: memory, register, condition-code, and CBAR. The memory implementation uses four memory locations; one location holds the barrier counter, and each thread has its own location on which to spin. Upon reaching the barrier, each thread performs a fetch and increment on the counter, using the MAP's memory synchronization bits to atomically lock and unlock the counter. If the barrier count is less than 2, the thread begins spinning on its own memory location. Otherwise, the other threads have reached the barrier, the count is reset to zero, and the spinning threads' memory locations are marked full, releasing them.

Barrier Method	Latency
Memory (cache hit)	61
Register	6
Condition Code	5
CBAR	1

Table 5: Latency to execute a barrier across all three clusters. Even with an on-chip cache, synchronizing using memory is more than ten times as expensive as using registers or the `cbar` instruction.

The register barrier microbenchmark consists of an even phase barrier, followed by an odd phase barrier. Upon reaching the barrier in an even phase, a thread empties its odd phase registers, and writes into the even phase registers of both of its neighbor threads. It then reads from its own even phase registers, stalling until they have been written by the neighbors. Two registers per phase are necessary to allow each of the neighbors to communicate independently. The Condition Code barrier is similar except that with the broadcast condition code registers, only one instruction is required to signal to both neighbor threads. The CBAR barrier uses the `cbar` instruction, without requiring any registers or auxiliary instructions to be executed.

Each mechanism is implemented in a simple program that does 100 successive barriers. The time per barrier in the steady state is measured and shown in Table 5. The `cbar` instruction is the fastest and can complete a barrier every cycle. The register and condition code barriers are similar, with Condition Code being one cycle faster since only one write is necessary to communicate with both neighbors. The memory barrier requires 61 cycles, even with all accesses hitting in the cache. For each thread, approximately 20 cycles are needed for the control overhead of testing the barrier counter, while the remaining cycles are consumed contending for the on-chip cache and waiting for the other threads to arrive at the barrier. In order to exploit fine–grain parallelism with task lengths in the 10s of cycles, long latency memory-based barriers cannot be used.

3.4 Synthetic Benchmark

A synthetic benchmark was developed to further examine the effect of the interthread register and memory communication latencies of the MAP chip. With fast mechanisms for thread invocation and communication, extremely fine–grain thread parallelism can be exploited. If slower mechanisms are employed, such as communicating using on-chip memory, fine–grain parallelism can still be exploited, but the granularity of the tasks must be larger.

The synthetic benchmark, shown schematically in Figure 4, consists of a single loop containing three function calls, each of which may be run in parallel. Varying `sub_num` changes the time to execute each of the function calls (affecting both grain size and problem size), and the number of outer loop iterations is dictated by `global_num`.

In the parallel versions, the master thread invokes one instance of `sub_loop` on each of the neighboring clusters, using a parallel procedure call (PPC), and executes the third instance itself. The slave threads operate in standby mode waiting to be signalled by the master. When a slave completes, it returns its result to the master, which performs a join before beginning another iteration of the outer loop. The versions to be compared are enumerated in Table 6.

```
for(i=0; i<global_num; i++) {
  res1 = sub_loop(sub_num, const);
  res2 = sub_loop(sub_num, const);
  res3 = sub_loop(sub_num, const);
  total_res = res1 + res2 + res3;
}
```

Figure 4: Pseudocode for synthetic benchmark. Each instance of `sub_loop` is executed on a different cluster for the parallel measurements.

Synthetic Program	Description
SEQ	Baseline sequential
PPC_REG	Parallel with register synchronization
PPC_MEM	Parallel with memory synchronization

Table 6: Synthetic benchmarks.

Figure 5 shows the time for one iteration of the outer loop as a function of the granularity of the inner loops, normalized to the sequential execution time. The granularity, in turn, is a function of the number of inner loop iterations, which is varied from 0 to 30. When no iterations are executed within `sub_loop`, the procedure call overhead and test inside the procedure still requires 19 cycles. Each increment in grain size corresponds to an additional loop iteration in each subroutine. At the smallest grain size, PPC_REG is 1.6 times faster than SEQ, while PPC_MEM is 1.2 times slower, due to the additional cost for the master to store the arguments into memory and the slave to retrieve them. Both PPC_REG and PPC_MEM improve substantially as more work is done inside the inner loops, but their execution time relative to sequential flattens out above granularities of 110 cycles as they approach the maximum of 3 times speedup. PPC_REG still maintains an advantage over PPC_MEM, but that diminishes as the granularity increases.

The most significant component of the overhead is in starting the slave threads. As shown in Figure 6, the cost to pass each additional argument from the master to a slave is not substantial. For PPC_REG, approximately two cycles are required for each additional argument, one cycle for each slave thread. PPC_MEM requires almost four cycles per additional argument, two cycles for each slave to perform an address calculation and a store.

4 Macro Applications

As shown with the microbenchmarks, the communication and synchronization mechanisms of the MAP chip allow threads to be invoked quickly and communicate efficiently with one another. This section explores the utility of these mechanisms in applications using inner–loop and outer–loop parallelism. Inner–loop parallelism is discovered by examining the inner loops of the applications to find subroutines and expressions that can be executed concurrently. Outer–loop parallelism comes from the outer loops of the applications, mainly by dividing the data set across the processors and

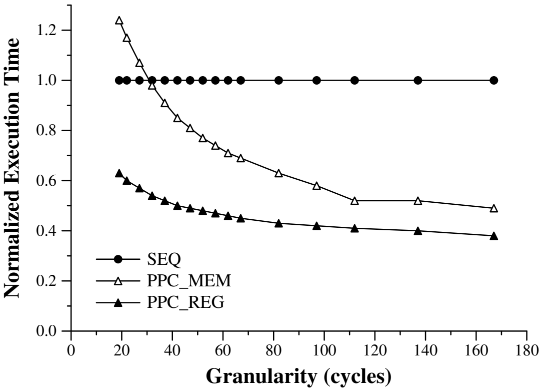

Figure 5: Outer loop iteration time as a function of inner loop grain size, normalized to sequential. At the smallest grain size (19 cycles of work in slave threads) PPC_REG is 1.6 times faster than SEQ. PPC_MEM becomes faster than SEQ at grain sizes of greater than 30 cycles.

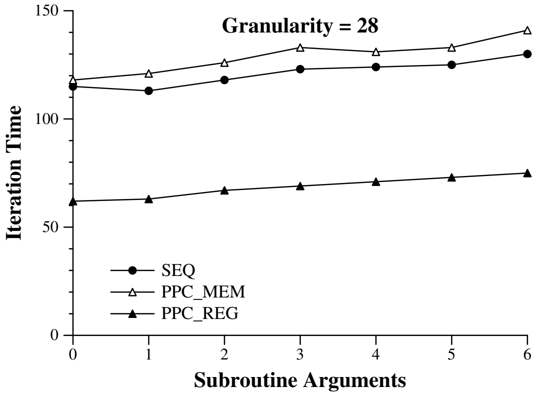

Figure 6: Outer loop iteration time as a function of the number of arguments passed from master to slave with a grain size of 27 cycles. PPC_REG requires two additional cycles per argument, one cycle for each slave thread. PPC_MEM requires almost four cycles per additional argument, two cycles for each slave.

assigning independent loop iterations to them. The experiments in this section show that inner–loop parallelism exploits concurrency in different parts of the program than outer–loop parallelism, and that the granularity of the inner–loop tasks is substantially smaller than outer–loop tasks.

4.1 Benchmarks

The applications in this study are compiled using MMCC, the MAP C compiler, a derivative of the Multiflow C compiler [9]. The compiler is able to compile a sequential program across all three arithmetic clusters. However, for the experiments reported in this paper, MMCC produces sequential single cluster code, using all three execution units within a cluster as a 3 instruction wide statically scheduled machine. MARS, the runtime system for the M-Machine, is used to provide system services, including memory allocation, terminal I/O, and file I/O [7]. While both MARS and the MAP support virtual memory, all experiments were run in a physical address space, with no TLB miss handling required.

Outer-loop parallelism is explicit in the applications and exploits concurrency at outer loops with data dependent phases separated by barriers. Inner-loop parallelism is implemented by encapsulating independent expressions and function calls inside procedures. The applications are detailed below and summarized in Table 7.

FFT solves a 1-dimensional partial differential equation using forward and inverse FFTs. With outer-loop parallelism, each processor is assigned a subsection of the array, and computes one level of the butterfly on its subarray before placing its result into a temporary array. After a barrier, each processor copies its section of the temporary array to the global array and barriers again. Inner-loop parallelism is extracted by executing inner-loop expressions and subroutines concurrently. The size of the array is varied from 4 to 128 complex numbers.

EM3D simulates electromagnetic interactions and consists of alternating phases of computation on `e-nodes` and `h-nodes`. To exploit outer–loop parallelism, each processor is assigned a subset of the nodes and at each timestep computes new values for its `e-nodes`, barriers, computes new values for its `h-nodes`, and barriers again. Inner–loop parallelism is exploited by computing all of the interactions for a given node concurrently. The EM3D initialization routines are not included in any results. The problem size is varied from 6 `e-node`/`h-node` pairs to 30 pairs, and each node is connected to 5 other nodes.

MG is a solution to a 3D Poisson partial differential equation and is based on the multigrid kernel from the NAS parallel benchmarks and SPEC95. The outer–loop parallel code assigns a subset of the three dimensional data space to each processor, and the different computation phases are separated by barriers. The inner–loop version parallelizes only the innermost loop of the `Relax` (relaxation) subroutine. Each of these inner loop iterations consists of a weighted sum of different array elements. These operations are forked to other clusters and the results are combined by the master before the next iteration begins. The volume of the cubic space to be solved is varied from 64 to 2744 double precision floating-point numbers.

CG implements a Modified Incomplete Cholesky Conjugate Gradient method for 3-D boundary value problems. The outer–loop parallelism profile forms a wavefront across the central diagonal of a cube that forms the problem space. At each iteration, a processor computes its assigned portion of the wavefront, and then executes a barrier. The inner–loop version only parallelizes the innermost computation loop which consists of a set of arithmetic operations combined with boundary checks to handle corners, edges, and faces of the cube. The volume of the cube is varied from 27 to 1728 double precision floating-point numbers.

EAR is from the SPEC92 suite and simulates the propagation of sound in the human cochlea (inner ear). The application consists of a sequential outer loop, containing sequences of parallel inner loops. Only the inner loops are parallelized as no outer-loop parallelism is available. Ten time steps are simulated and the size of the input vector is varied from 10 to 100 double precision floating-point numbers.

Benchmark	Source	Problem Size
FFT	Alewife [3]	4–128 complex doubles
EM3D	UC Berkeley [4]	6–30 node pairs
MG	Alewife [3]	64–2744 doubles
CG	Yeung [18]	27–1728 doubles
EAR	Spec92 [15]	10–100 doubles

Table 7: Benchmark Summary.

4.2 Inner–loop Parallelism

In each of the benchmarks, inner–loop parallelism is exploited by manually identifying independent expressions, function calls, and loop iterations inside an application's inner loop. A master thread forks the parallel work to a free cluster at runtime via a parallel procedure call (PPC) and waits only when the return value is needed, similar to a `future` [8]. To reduce overhead, assembly inlining in the compiled code is used to execute the special instructions needed to invoke a thread on a remote cluster, and to synchronize when complete. To evaluate the utility of the register interaction mechanisms, the arguments passed from the master to the slave using either registers or the on-chip cache. When using registers to communicate between the master and a slave, the master thread reserves and empties a register for the return value prior to forking the slave. At the join, the master stalls on the empty register until the slave writes into it. A slave empties its registers and stalls on them until the master delivers the function arguments. When using the on-chip cache, the memory synchronization bits are used to signal between master and slave to indicate that data is being transferred between them. The consumer spins on a memory location until the producer writes the data.

Figure 7 shows the inner-loop task granularity for all five applications as a function of problem size on a log–log plot. The granularity for inner-loop parallelism is defined as the average time for the slave threads to execute their parallel tasks. The problem sizes are indicative of the relative amount work for each benchmark, but cannot be compared across different applications. FFT, EM3D, and CG all exploit parallel expressions within their inner loops. Thus the granularity remains constant and small (less than 300 cycles) across the different problem sizes. EAR and MG each have inner loops that can be parallelized. As the problem size increases, so does the amount of work per parallel task.

Figure 8 shows the execution time for FFT across all of the problem sizes, normalized to the sequential execution time on a single MAP cluster. The **Cache** line shows the relative execution time when using the on-chip cache to communicate between the master and the slave threads, while **Register** uses the MAP's register communication mechanisms. **Optimal** is a measure of the execution time if all of the communication between the master and slaves occurs instantaneously. All three versions of the application improve relative to the sequential code as the problem size increases, with a 1.5 times speedup for **Register** at the largest data set. This is due to the application spending more time in the parallel section of the code relative to the sequential sections. Register communication is approximately 20% faster than using the on-chip cache for all problem sizes. However, the speedup of using multiple clusters is limited by the amount of parallelism

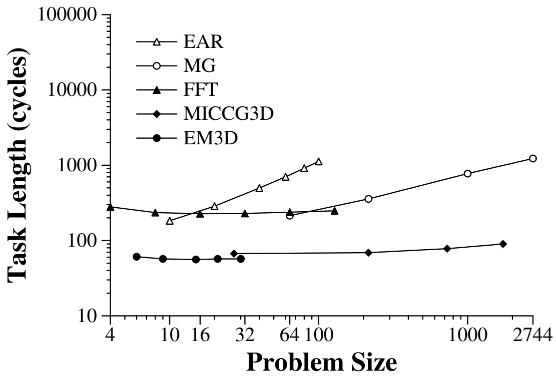

Figure 7: Inner–loop task length versus problem size. The task length is the average time for the slaves to execute their parallel tasks. FFT, CG, and EM3D exploit expression oriented parallelism in the inner loop, with granularity independent of problem size. EAR and MG exploit inner loop level parallelism and have granularities that increase with problem size.

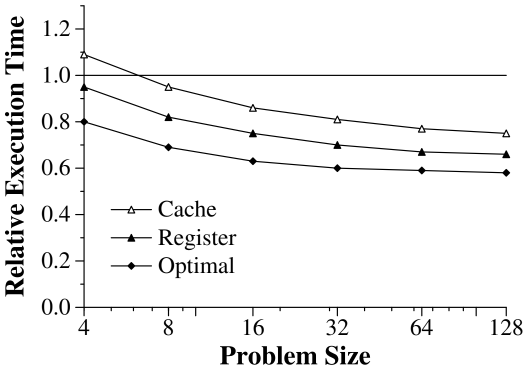

Figure 8: Normalized execution time versus problem size for Inner-loop FFT. The higher interaction latencies of **Cache** cause it to be consistently 20% slower than **Register**.

in the application and the method of extracting it, rather than by the communication overhead. Even when communication is free (**Optimal**), only an additional 15% of performance improvement is attained. The speedup for FFT is minimal at small problems sizes and improves as the size of the data set increases. With a 4 element input vector, FFT executes only 6 iterations of its inner loop, and the total execution time is dominated by the sequential component of the application.

Figure 9 illustrates these limitations by decomposing the running time of FFT with a problem size of 128 into execution and overhead components. The cycle breakdown is shown for a single cluster (**SEQ**) as well as the parallel versions using the on-chip cache and registers for communication. For the parallel versions, both the master (**M**) and two slaves (**S1**, **S2**) are shown. The primary factor that limits the overall speedup is the load imbalance seen in the parallel versions, as there is significant sequential work performed only by the master. The communication overhead using registers is less than one half that of using the cache, but the overall impact on performance is only 20%.

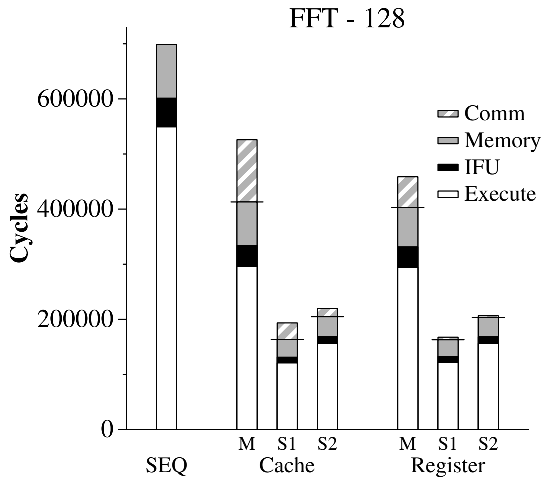

Figure 9: Cycle breakdown of inner–loop FFT with 128 word input vector. The execution time is broken down into the cycles spent executing instructions (**Execute**), waiting for the instruction fetch unit and instruction cache (**IFU**), waiting for data from the memory system (**Memory**), and communicating between the clusters (**Comm**).

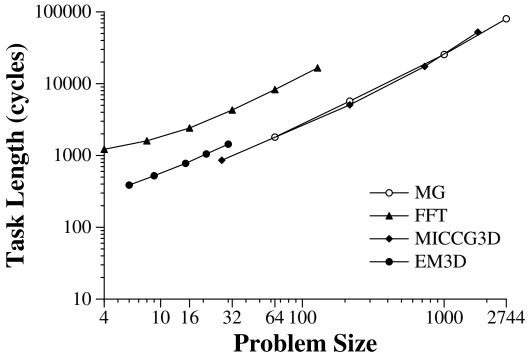

Figure 10: Outer–loop task length versus problem size. The task length is the average time between barriers. The outer-loop parallel tasks are much larger than inner-loop and increase dramatically with data set size.

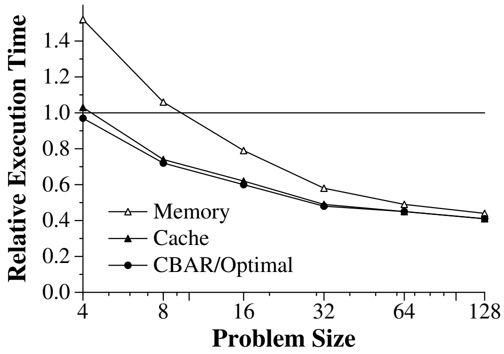

Figure 11: Normalized execution time versus problem size for Outer-loop FFT. As problem size increases, the difference between synchronizing via off-chip memory (**Memory**), the on-chip cache (**Cache**) and the barrier instruction (**CBAR**) diminishes.

4.3 Outer–loop Parallelism

Outer–loop parallelism is exploited using the shared-memory multiprocessor parallelizations of each of the applications, in which outer parallel loops are identified and executed concurrently on each of the three MAP clusters. The three clusters communicate using the shared memory system and can synchronize either through memory, or using the `cbar` instruction. Figure 10 shows the outer–loop task granularity on the same scale as the inner–loop granularity of Figure 7. Outer–loop task granularity is defined as the number of cycles spent between barriers. The gap in grain size between the inner and outer loop parallelizations is more than a factor of 10 for EM3D, MG, and CG, even on the smallest problem size, and it widens to a factor of 550 at a problem size of 1728 for CG. FFT exhibits the narrowest range, with a factor of 6 at vector length 4, to a factor of 70 at vector length 128. The large task lengths of the coarse grained applications stem from their original implementation on a shared memory multiprocessor, with interaction latencies in the thousands of cycles. Exploiting parallelism in the 80-200 cycle range would be infeasible with such high interaction costs.

The effect of this increasing granularity can be seen in Figure 11, which shows the execution time of FFT as a function of problem size, normalized to the sequential execution time. **Cache** shows the execution time when the barrier is implemented using the on-chip cache, while **CBAR** shows the execution time when the barrier instruction is used. **CBAR** is equivalent to an optimal barrier since the `cbar` instruction is so efficient. Outer–loop parallelism results in shorter execution times than inner–loop, as more of the code is parallelized and the larger grain size requires less communication and synchronization. FFT improves from no speedup on a 4 element vector to 2.4 times speedup on a 128 element vector. The improvement in speedup is a direct result of both the increasing granularity and the larger fraction of time spent in the parallel sections as the problem size increases. Another consequence of the coarse granularity is that the performance of the fast barrier **CBAR** and the memory barrier **Cache** are practically indistinguishable. Since so much time is spent between synchronizations, the cost of the barrier is inconsequential.

The coarse grained applications see substantial speedups on relatively small problem sizes for two reasons. First, synchronization cost is low, even using memory locks, because all of the accesses are local. Second, all of the data for the threads is shared either in the on-chip cache or in local memory. However, in a traditional multiprocessor, the communication costs are significantly higher. Inter-node barriers are more expensive and any shared data must be passed from node to node. The **Memory** curve in Figure 11 is intended capture some of the effect of additional synchronization cost by increasing the barrier overhead to 1000 cycles.

4.4 Summary

Figure 12 summarizes the execution time for all 5 benchmarks. The applications can be partitioned based on their task granularity into fine, medium, and coarse grain. On the MAP chip, fine–grain tasks are typically less than 300 cycles, medium grain tasks are

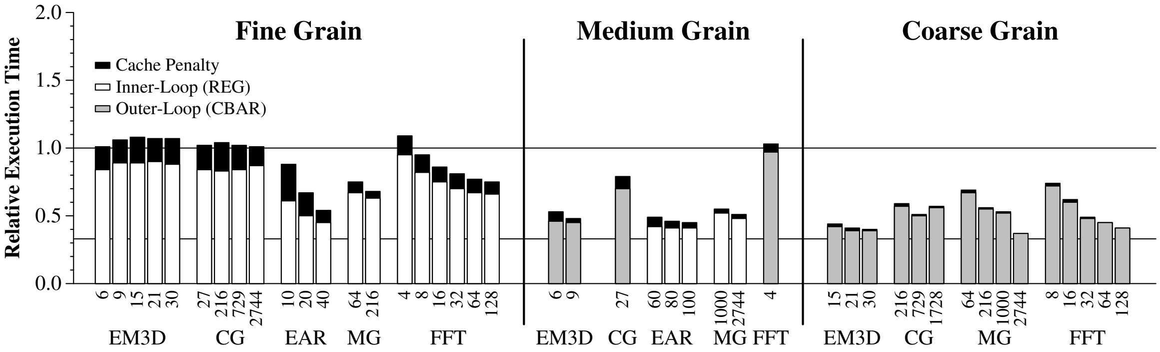

Figure 12: Normalized execution time of all 5 applications, including inner and outer–loop parallelization, across all problem sizes. The penalties for interacting using on-chip cache can be substantial, depending on the task granularity.

between 300 and 1500 cycles, and coarse grain tasks are greater than 1500 cycles. The task granularity is a function of the method of parallelization (inner–loop versus outer–loop), as well as problem size. The dark caps on the execution time bars signify the penalty for using the on-chip cache instead of the integrated communication and synchronization mechanisms of the MAP chip. As is evident from the graph, in order to exploit fine-grain tasks, the integrated mechanisms are a necessity. Medium grain tasks can be exploited using only the on-chip cache for communication and synchronization. Coarse grain tasks require no special mechanisms for synchronization since interaction frequency is small.

When outer-loop parallelism is available, it generally yields faster execution times than inner-loop parallelism, as demonstrated in Figure 13. However, some applications such as EAR have no outer-loop parallelism and require additional hardware support for communication and synchronization to improve performance. In addition, since inner and outer loop parallelism exploit concurrency in different components of the program, they can be used in concert to further improve application performance.

The experiments in this section demonstrate that there is considerable fine–grain thread parallelism in typical applications and that register-based communication and synchronization provides sufficiently low overhead to exploit this parallelism efficiently. The MAP's fast interaction mechanisms (10 cycle thread invocation, 1 cycle communication and synchronization) enable application speedups of up to 2.1 on three processors, using only inner–loop parallelism. The granularity of this fine-thread parallelism is typically between 80 and 200 instructions and is largely independent of problem size. Conventional multiprocessor mechanisms with long interaction latencies are unable to exploit fine threads at all. The coarse–thread parallelism that can be exploited in multiprocessors has a granularity of 10^3 to 10^5 instructions and is strongly dependent on problem size. Based on examination of the code, we expect that fine–thread parallelism will continue to scale with more processors and that more aggressive parallelization can yield both greater concurrency and smaller grain sizes.

5 Related Work

The study of synchronization cost performed in [2] explored a spectrum of granularities including instruction, statement, and loop level parallelism. They found that statement oriented parallelism was far more sensitive to synchronization overhead than loop level parallelism. However, even with substantial synchronization overhead the statement level parallelism still yielded 4 to 20 times speedup over sequential. This study suggests that the amount of fine-thread parallelism available in applications is considerably greater than what we have exploited so far using simple approaches to parallelization, and that it scales well beyond three processors. It also shows, as we have, that to extract this parallelism requires very low-overhead synchronization.

Architectures that support fine–grain threads in a multiprocessor typically implement fast thread creation and dispatch mechanisms. The *T architecture, whose threads are in the range of 15 instructions, implements `fork`, `join`, and `next` instructions that interact with a memory task queue, and a synchronization coprocessor to allow threads on different processors to communicate with one another [11]. Like the MAP chip, the Tera Computer System [1] also exploits fine-grain threads using a multithreaded multiprocessor architecture. In a Tera machine, interaction between threads takes place only through memory, and full/empty bits are provided on each memory location to enable fast synchronization. Tera's architecture also penalizes single threaded code as it has no support for data locality and uses a hardware scheduling policy which prohibits a single thread from using the execution resources on every cycle.

The Hydra and Simultaneous Multithreading (SMT) architectures also aim to scale on-chip parallelism beyond the limits of ILP. The Hydra architecture explores the design tradeoffs of building a single-chip multiprocessor, focusing on the memory system [10]. Coarse grained tasks execute independently and communicate via a level-1 or level-2 cache. SMT adds multithreading to a traditional superscalar to exploit both instruction and thread level parallelism [16]. Execution resources are dynamically assigned to different threads, and instructions from them may execute simultaneously. Both Hydra and SMT provide only memory-based mechanisms for communication and synchronization between threads and are thus limited to using relatively coarse-grain threads. Our work is complementary to these projects in that register-based mechanisms could easily be incorporated into these architectures, extending the granularity of parallelism they are able to exploit.

The Multiscalar architecture attempts to deduce fine–grain parallelism at runtime [14]. Basic blocks of the program are assigned dynamically to different execution units and hardware is responsi-

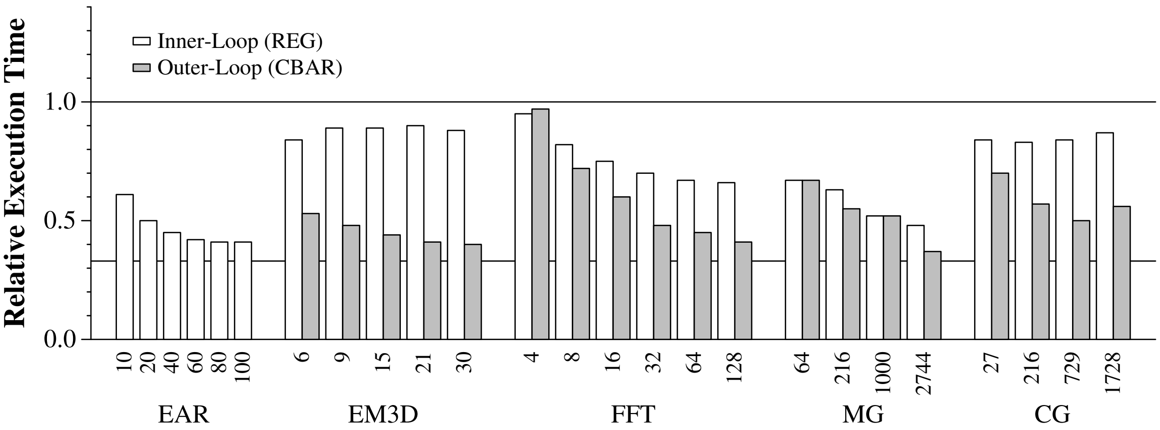

Figure 13: Normalized execution time for all applications comparing inner to outer loop parallelization.

ble for enforcing the data dependencies among the blocks. Communication takes place via a unidirectional ring to which each thread can read or write. This promising approach to extracting speculative fine-thread parallelism is well matched to implementations using register-based mechanisms in lieu of the special hardware suggested in [14].

The Cray X-MP implemented two central processing units with a bank of shared address, scalar data, and semaphore registers that could be accessed by either processor [12]. These registers were typically used for self scheduling of loops. The registers were not general purpose and values were copied to a processor's local register set prior to using the data.

6 Conclusions and Future Work

Instruction-level parallelism (1 cycle tasks) and coarse-grained concurrency (10,000 cycle tasks) dominate today's parallelism landscape. The more than 4 orders of magnitude between their granularities expose a parallelism gap that can be filled with 100 cycle tasks. However, the development of fine–grain programs has been a chicken–and–egg proposition. Fine–grain applications are not prevalent because there are no machines with fine–grain mechanisms, and vice versa.

The MAP chip architecture and silicon implementation introduce fast on-chip interprocessor mechanisms such 10 cycle thread invocation, 1 cycle communication latency, and a single cycle barrier instruction, Microbenchmark studies show that these mechanisms allow communication that is 10 times faster and synchronization that is 60 times faster than mechanisms that use an on-chip cache. With these low overhead operations, tasks of less than 100 cycles are now feasible. In the MAP chip, the cost of these mechanisms is small, as they are implemented by augmenting the existing cluster to memory communication paths.

Exploiting fine–grain thread parallelism using register-based mechanisms is also well matched to the wire-limited nature of future semiconductor processes. As technology advances, gate delay decreases but wire delay increases [13]. By 2007, forty 500ps clock cycles are expected to be needed to send a signal across the diagonal of a single chip. This trend motivates architectures that minimize global communication and the large latencies they imply. Structuring a future microprocessor as a number of superscalar processors that communicate and synchronize via registers keeps most communication local to individual processors. Global communication is made explicit in the processor microarchitecture allowing advanced circuit designs to target these long wires without affecting the processor's design. This partitioning will be even more useful as communication and synchronization can be pipelined to permit scaling to large numbers of on-chip processors.

In this study, the MAP's fast communication mechanisms are used to implement a parallel procedure call (PPC), in which a master thread dynamically assigns work to the slave threads on the other execution units. Parallelizing the inner loops of several applications using PPC yields performance improvements of 1.2–2.1 times even on small problem sizes. The register communication mechanisms result in a 20% improvement over communication via the on-chip cache. The measured speedup is limited by both the overhead of thread control, and by the sequential components of the program which are not accelerated.

For the last 15 years, single microprocessor performance has increased by 50% per year with about half the improvement coming from faster devices and the rest due to increased parallelism. Today's 4–8 issue superscalar processors are nearing the limits of ILP. To remain on this performance curve, parallelism beyond ILP must be exploited on a single chip. Fine–grain thread parallelism is well suited to fill this performance gap, and well matched to the cluster organizations of future microprocessors. Most applications, even those with small problem sizes, have considerable fine-thread parallelism, and this parallelism, because of its limited extent, has a smaller cache footprint than coarse-thread alternatives [6].

Discovering fine–grain parallelism in expression oriented programs is a major challenge. Aside from hand parallelization, compilers may be able to analyze and partition inner loop iterations, procedure calls, and expressions. Other avenues, such as pipelining dependent loop iterations across the on-chip processors, or speculatively executing components of the program in parallel are possible as well. Regardless of the technique, fine–grain threads enable a different and orthogonal type of parallelism than that found in outer loops. Reducing the synchronization and communication costs between parallel tasks will enable fine–grain parallelization of programs, and allow existing problems, such as personal or business applications, to be solved faster without scaling their size.

Acknowledgements

Many thanks to the anonymous reviewers of this paper for their insightful suggestions, and to Sun Microsystems for their generous equipment donations. Thanks also to the Spectrum Design Services group at Cadence Design Systems for their contributions to the physical design of the MAP chip.

References

[1] ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. The Tera computer system. In *Proceedings of the International Conference on Supercomputing* (June 1990), pp. 1–6.

[2] CHEN, D.-K., SU, H.-M., AND YEW, P.-C. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture* (May 1990), pp. 239–248.

[3] CHONG, F. T., LIM, B.-H., BIANCHINI, R., KUBIATOWICZ, J., AND AGARWAL, A. Application performance on the MIT alewife machine. *IEEE Computer 29*, 12 (December 1996), 57–64.

[4] CULLER, D. E., DUSSEAU, A., GOLDSTEIN, S. C., KRISHNAMURTHY, A., LUMETTA, S., VON EIKEN, T., AND YELICK, K. Parallel programming in Split-C. In *Supercomputing* (November 1993), pp. 262–273.

[5] FILLO, M., KECKLER, S. W., DALLY, W. J., CARTER, N. P., CHANG, A., GUREVICH, Y., AND LEE, W. S. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture* (Ann Arbor, MI, December 1995), ACM, pp. 146–156.

[6] FISKE, S., AND DALLY, W. J. Thread prioritization: A thread scheduling mechanism for multiple-context parallel processors. In *Proceedings of the First IEEE Symposium on High-Performance Computer Architecture* (Raleigh, NC, January 1995), pp. 210–221.

[7] GUREVICH, Y. The M-Machine operating system. Master of Engineering Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1995.

[8] KRANZ, D. A., HALSTEAD, R. H., AND MOHR, E. Mul-T: A high-performance Lisp. In *Sigplan '89 Symposium on Programming Language Design and Implementation* (June 1989), pp. 1–10.

[9] LOWNEY, P. G., FREUDENBERGER, S. G., KARZES, T. J., LICHTENSTEIN, W. D., NIX, R. P., O'DONNELL, J. S., AND RUTTENBERG, J. C. The multiflow trace scheduling compiler. *The Journal of Supercomputing 7*, 1-2 (May 1993), 51–142.

[10] NAYFEH, B. A., HAMMOND, L., AND OLUKOTUN, K. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture* (May 1996), pp. 67–77.

[11] NIKHIL, R. S., PAPADOPOULOS, G. M., AND ARVIND. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer Architecture* (May 1992), pp. 156–167.

[12] ROBBINS, K. A., AND ROBBINS, S. *The Cray X-MP/Model 24*. Springer-Verlag, 1987.

[13] The national technology roadmap for semiconductors. Semiconductor Industry Association, 1997.

[14] SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. Multiscalar processors. In *Proceedings of the 22nd International Symposium On Computer Architecture* (May 1995), pp. 414–425.

[15] Spec benchmark release v1.1, 1992.

[16] TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium On Computer Architecture* (May 1995), pp. 392–403.

[17] WALL, D. W. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (1991), ACM, pp. 176–188.

[18] YEUNG, D., AND AGARWAL, A. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May 1993), pp. 187–197.