Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine[1]

David E. Culler, Anurag Sah, Klaus Erik Schauser
Thorsten von Eicken, John Wawrzynek

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California, Berkeley
Berkeley, CA 94720

Abstract: In this paper, we present a relatively primitive execution model for fine-grain parallelism, in which all synchronization, scheduling, and storage management is explicit and under compiler control. This is defined by a threaded abstract machine (TAM) with a multilevel scheduling hierarchy. Considerable temporal locality of logically related threads is demonstrated, providing an avenue for effective register use under quasi-dynamic scheduling.

A prototype TAM instruction set, TL0, has been developed, along with a translator to a variety of existing sequential and parallel machines. Compilation of Id, an extended functional language requiring fine-grain synchronization, under this model yields performance approaching that of conventional languages on current uniprocessors.

Measurements suggest that the net cost of synchronization on conventional multiprocessors can be reduced to within a small factor of that on machines with elaborate hardware support, such as proposed dataflow architectures. This brings into question whether tolerance to latency and inexpensive synchronization require specific hardware support or merely an appropriate compilation strategy and program representation.

1 Introduction

Multithreading at the instruction level may provide the key to general purpose parallel computing[26], because it allows the processor to tolerate long, unpredictable communication latency [2, 4, 17, 24, 29]. In addition, this level of multithreading is required to support certain modern parallel programming languages[28], such as Id[20] and Multilisp[18], and extensions of more conventional languages with synchronizing data structures, *e.g.* I-structures[6]. On the other hand, asynchronous transfer of control (context switching) is notoriously expensive on current machines, leading many researchers to examine asynchronous parallel execution models through the study of real machines[11, 13, 15, 22, 25, 27], paper architectures[1, 5, 14, 16, 19], and abstract machines[21]. In all of these proposals, the scheduling of threads is viewed as a property of the machine, invisible to the compiler. While we share the view that asynchronous events are the rule, not the exception, in large-scale multiprocessors, we claim that relieving the compiler of responsibility for scheduling low-level program entities squanders critical processor resources, such as high-speed register storage, and places unreasonable demands on the hardware, such as maintaining scheduling queues of

[1]A version of this report is to appear in the *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

Work described here was supported in part by the National Science Foundation under Presidential Young Investigator Awards CCR-905834 and MIP-8958568. Computational resources were provided in part under NSF Infrastructure Grant CDA-8722788. K. E. Schauser received support from the International Computer Science Institute. T. von Eicken is supported through the Semiconductor Research Corporation.

arbitrary size. By retaining some of this control, the compiler can optimize the use of processor resources for the expected case, rather than the worst case, and exploit considerable inter-thread execution locality. Thus, tolerance to latency and inexpensive synchronization require that the compiler adopt a suitable program representation, but this need not be manifest in the processor architecture.

To investigate this view, we have formulated a relatively primitive threaded abstract machine (TAM) in which processor resources and thread scheduling are explicit at the instruction level and all storage management is the responsibility of the language system, not the machine. We have retargeted the compiler for Id, an extended functional language that relies on dynamic scheduling, to generate TL0 (Thread Language Zero), a first-cut instruction set for this machine, rather than dataflow graphs. Although TAM could be realized directly in hardware, it is intended as a vehicle for studying what architectural support is most important for full-scale parallel programs on large parallel machines. To this end, we have developed a versatile translator for TL0 to a variety of existing parallel and sequential machines.

Preliminary measurements indicate that this implementation of Id yields performance between C and Lisp for comparable programs on the same uniprocessor. This disspells the view that the implementation of languages with fine-grain synchronization on conventional architectures must be essentially interpretive. Secondly, dynamic instruction counts under this execution model are comparable to dataflow models, which support synchronization and generation of parallel threads as part of every instruction. Third, the locality among execution threads that are not, or cannot be, statically ordered appears to be substantial. Finally, a large fraction of the potential synchronization events can be compiled away or synthesized cheaply with little or no architectural support. The key architectural challenge is an intimate coupling of processor and network, with much of the message decode task delegated to the compiler.

The following describes TAM and its current realization in TL0. Section 2 outlines the basic structure and scheduling mechanism supported by our model and draws comparisons with proposed and existing threaded execution models. Section 3 provides preliminary performance measurements. Appendix A describes our threaded machine language.

2 The TAM Execution Model

2.1 Storage model and basic structure

TAM recognizes three major storage resources—code-blocks, frames and structures—and the existence of critical processor resources, such as registers. A program is represented by a collection of re-entrant *code-blocks*, corresponding roughly to individual functions or loop bodies in the high-level program text. A code-block comprises a collection of *threads*; each thread is a sequence of instructions. Invoking a code-block involves allocating a *frame*—much like a conventional call frame—depositing argument values into locations within the frame, and enabling threads within the code-block for execution. Figure 1 illustrates the relationship between the code-block and the frame. Instructions may refer to registers and to slots in the current frame; the compiler statically determines the frame size for each code-block and is responsible for correctly using slots and registers under all possible dynamic thread orderings. (This is somewhat more complex than traditional register allocation via graph coloring[7].) The compiler also reserves a portion of the frame as a *continuation vector*, used at run-time to hold pointers to enabled threads. The continuation vector must be large enough to describe the concurrently enabled threads for a code-block. The global scheduling pool is the set of frames that contain enabled threads.

Structures are heap allocated data objects, accessed through split-phase fetches and stores.

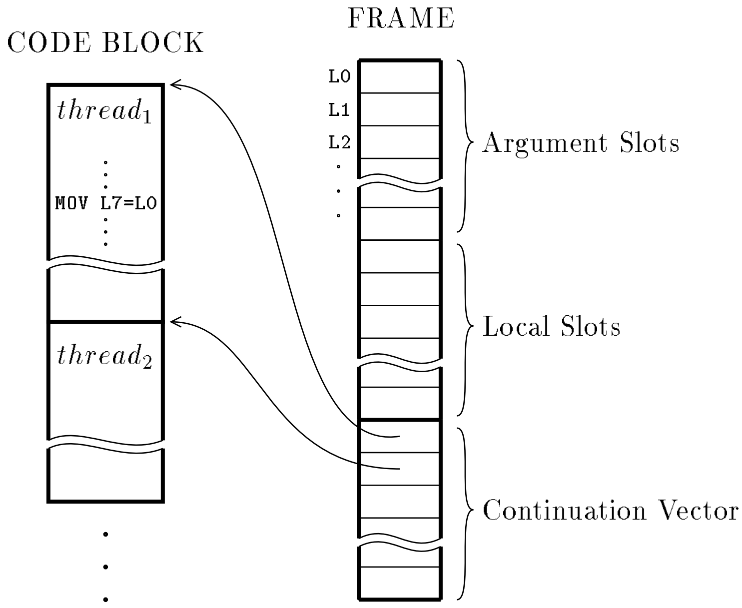

Figure 1: Basic Storage Resources. A code-block comprises a collection of *threads*; each a sequence of instructions. Invoking a code-block involves allocating a *frame* for local variables and for the continuation vector holding the list of enabled threads.

That is, the instruction that issues a fetch request does not wait for the data value to be returned, instead the response will initiate a new execution thread[4, 6]. This allows the processor to be well utilized while remote requests are outstanding. In addition, a synchronization event may take place at the site of the accessed object, so the request latency is unbounded.

2.2 Activations

An executing code-block may invoke several code-blocks concurrently, since the caller is not suspended as in a conventional sequential language. Therefore, the set of frames in existence at any time form a tree, the *activation tree*, rather than a stack, reflecting the dynamic call structure (see Figure 2). We refer to a frame and the set of threads executed relative to the frame as an *activation*. The basics of this parallel call scenario are well described in the literature[5, 9, 21]. To allow greater parallelism and to support languages with non-strict function call semantics,[2] the arguments to a code-block may be delivered asynchronously; each will initiate an execution thread within the code-block. An activation is *enabled* if its frame contains any enabled threads. At any time, a subset of the enabled activations may be *resident* on processors, as discussed below.

2.3 Threads

Threads come in two forms, *synchronizing* and *non-synchronizing*. A synchronizing thread specifies a frame slot containing the *entry count* for the thread. Each fork to a synchronizing thread causes

[2] In Id and other non-strict languages, it cannot be assumed that all arguments to a function can be computed before invoking the function. One or more of the arguments may, in fact, depend on a result of the function.

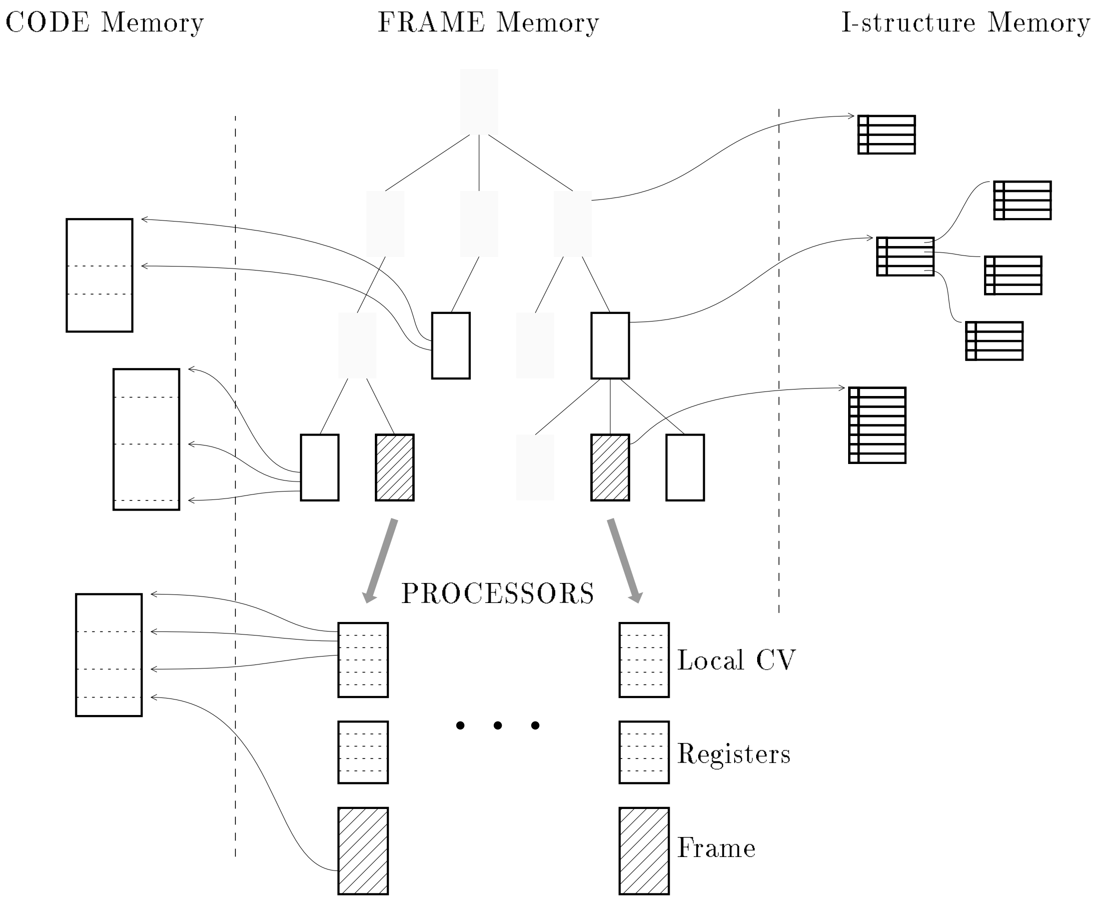

Figure 2: Basic Structure of an Executing TAM Program. An executing TAM program generates a tree of frame activations. The set of light-gray frames represent activations that contain no enabled threads, and thus are not included in the scheduling pool (not shown). The remaining activations are enabled. A subset of these, indicated by the cross-hatch are the ones that are currently resident on a processor. Only the resident activations can access processor registers and the local continuation vector.

the entry count to be decremented, but the thread executes only when the count reaches zero.[3] Synchronization occurs only at the start of a thread; once successfully initiated, a thread executes to completion. *Fork* operations may occur anywhere in a thread, causing additional threads to be enabled for execution. An enabled thread is identified by a continuation—its instruction pointer and its frame. Because the continuation vector is contained within the frame, a continuation is represented simply as a pointer to the first instruction of a thread.[4] A thread ends with an explicit *stop* instruction, which causes another enabled thread to be initiated, *i.e.*, removes an element from the current continuation vector and transfers control to it.

Conditional flow of execution is supported by *switch*, which forks one of two threads based on

[3] An implementation of the abstract machine may elect to associate the decrement with the fork, so a continuation is created only when the thread is enabled, or to simply create a continuation at the fork and perform the decrement as the first operation in the thread; the thread is cancelled if the counter is not zero. In either case, the decrement-and-test must be atomic relative to other threads.

[4] This means the abstract machine can be easily implemented with the word size equal to the address size. Many of the proposed threaded architectures require two addresses per word.

a boolean input value, and *case*, an indexed fork based on a small integer input. The compiler is responsible for establishing correct entry counts for synchronizing threads prior to any fork to the thread. This is facilitated by allowing a distinguished *initialization thread* in each code-block, which is the first thread executed in an activation of the code-block. One of the threads contains a *release* instruction that causes the frame to be reclaimed; the compiler ensures that this is the last instruction executed for the activation.

2.4 Quanta

Given that the execution model supports a tree of activations, many of which may have several concurrently enabled threads, the fundamental concerns are where frames reside in the storage hierarchy, how the pool of continuations is represented, and how threads are scheduled. Surprisingly, these concerns have received little attention in the threaded execution models discussed in the literature; TAM was developed to address these issues directly. The key observation is that the activation tree and the continuation pool are typically quite large, except on toy programs. This has been demonstrated empirically for programs in Id[8], Sisal[23], and Multilisp[18]. Minimizing the activation tree size while exposing sufficient parallelism is an active area of research, but even with advances in this area we cannot expect the entire activation tree or the entire continuation pool to be maintained in high-speed processor storage. Therefore, the scheduling mechanism must recognize that only a subset of the activation frames are resident on a processor and that a large number of continuations will exist for non-resident frames.

The storage hierarchy is explicit in TAM. In addition, scheduling is explicit and reflects the storage hierarchy. In order to execute threads from an activation, the activation must be made *resident*. Only a limited number of activations may be resident. When an activation is made resident on a processor, it has access to processor registers. Furthermore, it remains resident and executing until no enabled threads for the activation remain. The set of threads executed during a single residency is called a *quantum*. Recognition of this intermediate level of scheduling is a major departure from dataflow oriented execution models, such as ETS[22] and P-Risc[21], and is key to an efficient implementation on conventional machines.

A non-resident frame may accumulate several continuations, say as arguments are supplied, and when it becomes active all of these threads are executed, as well as any that they enable. Processor registers are essentially an extension to the frame with a lifetime of a single quantum. They may carry values between instructions within a thread or across threads in a quantum. All threads enabled by fork instructions are guaranteed to execute while the frame is resident. Therefore, the continuation vector is split into two parts: a remote continuation vector, used to hold continuations generated external to the activation, and a local continuation vector, used to hold internally generated continuations. The remote continuation vector is part of the frame whereas the local continuation vector should be viewed as a stack of continuation registers. Fork and stop translate into instruction pointer push and pop-jump respectively.

Quantum boundaries in TAM are visible to the compiler. When an activation is made resident, a distinguished initialization thread executes before any threads in the remote continuation vector. In the simplest case, the initialization thread for the first quantum of an activation will establish entry counts for the synchronizing threads and nullify the initialization thread for later quanta. Similarly, a distinguished completion thread is executed when the local continuation vector is otherwise empty. Again in the simplest case, this will extract a new enabled frame from the scheduling pool, make it resident, and transfer control to the initialization thread for the new quantum. This mechanism allows the compiler to control the use of processor registers. Frequently referenced frame slots may be "cached" in registers, with the initialization thread set up to restore them from the frame.

Values judged likely, but not proven, to have a lifetime of a single quantum may be kept in registers, with the boundary threads configured to save and restore them. (Our compiler does not yet exploit this capability, but we expect this quasi-dynamic scheduling will prove quite important in the long run.)

The representation of the scheduling pool is not specified by the model, but determined by the compiler. Sufficient space is provided in each frame to represent whatever data structure the compiler uses to organize the pool, *e.g.*, queue, distributed queue, priority tree, etc. The compiler may even elect to specialize the structure to reflect different scheduling policies in different portions of the program.

2.5 Inlets and split-phase operations

Thus far, we have focused on the interactions of threads internal to an activation. We now describe TAM facilities for inter-frame interactions, *i.e.*, passing arguments to an activation, returning results, allocating a remote resource (a frame or a structure), or accessing a remote structure. In each of these cases, the external entity needs to transmit one or more values back to the activation, cause them to be deposited in frame slots and cause a thread to be scheduled to indicate the arrival of the values. Handling of such a response is usually viewed as a machine primitive. For example, in P-Risc[21] an I-structure fetch instruction issues a request to the hardware module containing the location to be read, passing the frame pointer, slot number, and thread pointer. If the location is empty, the request is stored in the memory module until the location is written. When the location is full, the value it contains is sent back to the requesting processor, along with the frame, thread, and slot information. The hardware is expected to interpret this message, store the value in the specified slot and schedule the specified thread. On Monsoon[22], only a frame pointer and instruction pointer need be carried with the request, since the slot number is specified by the instruction. However, this relies on presence-bits associated with each frame slot.

TAM attempts to minimize the amount of information carried on such messages and to minimize the interpretation required upon their reception. To this end, each code-block has a set of *inlets*, in addition to a collection of threads. The inlets define its external interface. By convention, the low numbered inlets are used to receive argument values. The compiler generates an additional inlet for each value returned to the activation from a subordinate activation and for each reply to a split-phase request. Thus, the I-structure fetch instruction sends the frame pointer and inlet number (in addition to the address to be read) to the remote memory module. Each inlet is a simple sequence of instructions that extract components of the corresponding message, store values into slots of the specified frame and post threads in the corresponding continuation vector. In other words, an inlet is a specialized message handler for exactly one kind of message. It is like a thread, but extremely limited in its capability and may interrupt the currently active thread. An inlet can, however, handle messages of arbitrary length, network interface permitting.[5]

2.6 Comparison with other models

The basic structure of TAM is similar to several of the multithreaded architectures derived from dynamic dataflow, with some key differences. Iannucci's hybrid architecture[16] has a similar storage hierarchy; instructions may refer to processor registers or to slots in the current frame. However, the hybrid proposal associates presence bits with frame slots and when a thread attempts to read an empty slot it is *suspended*; the continuation for the thread is placed in the empty slot and

[5] Inlets provide a form of message handling similar to Dally's J-machine[10], but are more limited. The dispatch is trivial and storage required for the inlet to complete its task is pre-allocated in an activation frame.

rescheduled when the slot is written. Registers vanish at a point of potential suspension. To allow multiple references to frame slots, the hardware must support lists of suspended continuations.

Monsoon[22] associates presence bits with frame slots, but, when a thread attempts to read an empty slot the value carried on the token is written into the slot and further processing of the instruction is cancelled. The data is picked-up again when the instruction is re-enabled by another token, at which point the instruction executes and enables one or two further instructions. The addressing capabilities of Monsoon instructions are rather limited: only one frame slot and the data value carried on the current token can be referenced. P-Risc, which strongly influenced TAM, uses frame slots for synchronization, rather than presence bits, and makes the synchronization operation explicit. However P-Risc does not recognize a storage hierarchy (there are no registers) or scheduling hierarchy (the next thread may come from any frame).

Recent dataflow machines[12, 22, 25] allow several instructions, *i.e.*, a thread, to be enabled by a single dataflow synchronization. Registers are used to hold values with a lifetime of a single such thread, as these values need never be placed in the frame. However, these machines do not allow registers to carry values across threads, which severely restricts the usefulness of registers, as shown in Section 3. Also, the set of enabled threads is maintained in a special hardware token queue.

Several multithreaded architectures have been proposed as generalizations of conventional single-threaded machines, with registers sets (*i.e.*, frames) multiplexed to hide memory and communication latency[1, 14, 17, 27, 29]. In most cases, only one thread of execution per frame is supported. Thus, each outstanding reference has an entire register set standing idle behind it. With the exception of MASA[14], the number of frames per processor is static, thus the mechanism does not directly support language models with dynamically generated parallelism. By viewing memory as split-phase transactions, TAM allows multiple outstanding references per register set and minimizes the number of register set switches. Although TAM does not rely on multiple register sets in hardware, it could directly benefit from them.

2.7 Summary

To summarize the abstract machine, the program is represented by a collection of code-blocks. The state of the computation is described by the activation tree, the heap of structures, and the extended "processor state" of the resident activations. The scheduling pool consists of a distributed data structure containing those frames with non-empty continuation vectors. Processors execute threads from their resident activations as long as possible. The abstract machine provides four levels of synchronization of increasing cost and decreasing frequency: simple sequencing of instructions within a thread, scheduling of threads generated internal to an activation, scheduling of threads generated external to an activation, and scheduling of quanta. The first two are represented directly in TL0 instructions; the latter two are synthesized in inlets and boundary threads. When an activation becomes resident, as much work as possible is done on the activation, using the least expensive form of synchronization wherever possible. Thread-to-thread and activation-to-activation transitions are explicit in the model, so they can be controlled to a large extent by the compiler. The compiler produces specialized message handlers as inlets to each code-block. Appendix A describes TL0, a simple instruction set that embodies this model and provides a basis for comparison with other execution models.

3 Preliminary Measurements

To validate the TAM paradigm, a prototype back-end was developed for the MIT Id88 compiler that generates TL0 code, rather than dataflow graphs. A second, rather simple compiler trans-

Program	MMT 200	QS 5,000	QS 50,000	AS 1,500	AS' 1,500
Id	67 (57,31)	6.0	72	27	23
Lisp	441+51	2.4+10	27+414	55+0	36+0
C	15	0.5	8.0	2.6	1.5

Table 1: User Time Comparison (in seconds) of compiled Id, Lisp, and C on a MIPS R3000

lates the TL0 code to C, and a conventional C compiler is used for final machine code generation. This approach provides conservative execution times for a TAM-based implementation of a language requiring fine-grained dynamic synchronization. Timing measurements for a collection of small programs indicate that this implementation of Id is roughly competitive with conventional languages, *i.e.*, between C and Lisp on a single processor for similar programs. Further improvements will be realized in a more sophisticated version of the compiler, currently under development, by producing machine code directly and by more sophisticated generation of TL0. High quality compilation for conventional machines provides a baseline for assessing the performance of novel architectures with dynamic instruction scheduling.

In expanding the TL0 code to C, instrumentation code is inserted to collect TAM-level statistics. In this section, dynamic instruction mix measurements are compared with measurements of the MIT Tagged-Token Dataflow Architecture obtained with the Id World instruction-level emulator. This shows that TL0 instructions are essentially at the same level as TTDA instructions and demonstrates the benefits of compiling graphs into threads. Finally, dynamic TL0 instruction and scheduling data on large Id program runs is presented and used to derive the net cost of synchronization under this approach and with direct hardware support.

3.1 Performance of Id on stock hardware

Table 1 shows elapsed user time for several small programs written in Id, Lisp, and C and executed on a MIPS R3000. The Id programs are compiled to TL0, which is translated to C and compiled with optimization level "-O2". No inter-thread register usage is exploited. The Lisp programs are compiled using Allegro CL 3.1.0 (speed 3, safety 0). User time reported for the Lisp program is partitioned into program execution time and garbage collection time (separated by "+" in the table). The C programs are compiled with MIPS CC version 2.11 with optimization level "-O3". This data indicates that the net gain anticipated from hardware support for dynamic synchronization is considerably less than previously believed. Stock implementations need not execute at "interpreter speeds" if the compiler structures the representation of the program to minimize the cost of the most frequent forms of synchronization.

The data in Table 1 should be considered in light of the different implementation requirements of the languages. Id is a strongly typed, polymorphic language; thus no data tagging is required at run-time. However, the current version of the type system does not distinguish between integers and reals, so in the Id implementation all numbers are represented as 64-bit IEEE floats. Id version 90.0 remedies this situation. The TL0 code is translated into very unusual C, which defeats the optimization and register allocation heuristics of the C compiler. Direct translation to the host instruction set could yield considerable improvement. Common Lisp is dynamically typed and the implementation represents values as 32-bit tagged quantities, thus floats are accessed indirectly. In C, types are explicit, neither dynamic typing nor dynamic synchronization is required. Data structures in Id are non-strict, *i.e.*, the structure can be accessed before all its elements are defined, and require element by element synchronization. The current TL0 implementation uses an array of tag bytes to represent the status of an array of values.

MMT is a simple matrix operation test; it creates two double precision identity matrices, multiplies them, and subtracts a third identity matrix. Performance of the Id version is within a factor of five of the C program and substantially faster than the Lisp program. The C compiler comprehends the simple inner loop and unrolls it four-fold. These optimizations are not attempted on the C generated from TL0. The current version of the Id compiler generates many unnecessary moves that are easily eliminated. To simulate the compiler improvement and the impact of the more precise type system, we improved the inner loop by hand, bringing the time down to 57 seconds. Carrying values in registers across threads (at the TL0 level) and transforming `FORK-STOP` to a simple branch, brings this down to 31 seconds.

The next two columns show the user time for Quick-Sort on a list of random numbers, using accumulation lists. Here the relative difference with C is greater and Id is faster than Lisp only when GC time is included. The poorer performance of the Id version is due, in part, to our conservative thread partitioning in conditionals and the high frequency of calls, which currently involve a request to the UNIX *malloc* for activation frame storage.

The final two columns show user time on an array selection sort. The function used to compute the key is passed as a argument to the sort routine. The Id language system is designed to deal with higher order functions and is able to perform optimizations that would be difficult in the other languages. For the final column, we optimized these programs by hand; the Id version improves only slightly, while the other improve considerably.

3.2 Threads versus dataflow graphs

To demonstrate the impact of compiling to threads, Table 2 presents dynamic instruction frequencies for Id programs compiled to TL0 and compiled to graphs for the MIT TTDA[5]. In addition to the small programs discussed above, this includes two larger programs. Gamteb is a Monte Carlo neutron transport code. It is highly recursive with many conditionals. Simple is a hydrodynamics and heat conduction code widely used as an application benchmark, rewritten in Id[3]. The TTDA numbers were obtained using the Id-World graph interpreter, with the same suite of arithmetic operators, structure operations, and the same resource management operations as in TL0. In the instruction counts, the `STOP` terminating each thread is viewed as part of the `FORK`, `IFETCH`, `ALLOC`, or `RSEND` that enabled the thread. Similarly, the `SYNC` starting each synchronizing thread is not counted, since the decrement and test of the entry count is performed where the thread is spawned.

The *Ops* row shows the TL0 formulation performs roughly 50% more instructions than the TTDA. A single TTDA instruction can synchronize two arguments, compute a result, and generate several copies of the result. The token count measurements indicate that on average an instruction produces 1.5 output tokens (and consumes 1.5 input tokens). A direct thread-per-node transliteration to TL0 would add 1.5 fork operations to each TTDA instruction, resulting in an 150% increase in instruction count with the accounting scheme used for Table 2. Compilation to threads avoids a large fraction of this overhead, without compromising the non-strict semantics of the language.

The net reduction of synchronization events can be seen in the row labeled *aborts* which shows the number of unsatisfied entry count decrements, for the TL0 case, and match failures, for the TTDA, as a percentage of the total number of TTDA operations. The recent generation dataflow machines[12, 22, 25] all support some form of thread, and the graph partitioning and elimination of redundant arcs used in TL0 compilation could be employed with similar benefits.

The lower two sections of Table 2 show the instruction counts broken down into classes as a percentage of the number of TTDA instructions, *i.e.*, the TL0 entries are normalized to the TTDA

Dynamic Instruction Mix										
Run	as 1 100		gamteb 128		mmt 50		qs 512		simple 2 10	
	TL0	TTDA	TL0	TTDA	TL0	TTDA	TL0	TTDA	TL0	TTDA
Activations	105		6,653		2,757		3,075		4,619	
Ops	368,116	256,803	611,903	383,434	3,363,250	2,346,452	540,355	349,384	677,956	563,960
Threads	83,536		174,241		472,506		207,090		165,018	
Tokens		386,969		564,030		3,248,537		540,979		719,311
Aborts	15.8	50.7	22.7	47.1	6.4	38.4	21.7	54.8	20.2	27.5
Arith	38.0	30.2	25.3	25.5	51.1	39.9	13.6	13.4	33.5	39.2
Ifetch	10.0	10.0	10.3	9.9	10.9	10.9	6.7	6.6	11.5	13.4
Istore	0.1	0.1	4.1	4.0	0.4	0.4	3.72	3.71	1.6	1.8
Rsend/Tag	0.4	0.4	8.7	9.8	1.3	1.4	5.3	6.2	7.3	9.5
Alloc/Resource	0.1	0.1	1.8	3.0	0.2	0.2	3.0	3.8	1.0	2.1
Move/Identity	56.5	13.1	61.4	24.1	65.1	4.5	56.4	21.7	35.1	23.6
Fork&Switch	37.9	21.9	47.2	12.0	14.3	12.3	66.2	25.4	29.5	3.3
-/D Loop		24.2		10.7		30.0		19.4		5.2
Ratio	1.43 : 1		1.59 : 1		1.43 : 1		1.55 : 1		1.20 : 1	

Table 2: Instruction Frequency comparisons between TL0 and the MIT TTDA

counts. The middle section represents the essential computation; the counts should be roughly equal under the two execution models. (The difference in arithmetic counts on MMT is due to operations that convert the index used in structure operations to an integer, which is not required under the tagged execution model of the TTDA.) The lower section includes the control and data movement "overhead" operations, which differ in the two models. In the TAM control-flow paradigm, it is necessary to move values to locations where they will be accessed, for example, at the bottom of a loop or conditional. The current TL0 compiler introduces a large number of unnecessary moves, which will be remedied in the next version. In the dataflow paradigm, values are delivered directly to the instructions that access them. Split-phase operations on the TTDA are limited to producing a single result, so an *identity* instruction is introduced to fan-out the result where needed. With the fan-out of nodes in the dataflow graph limited to two, as on Monsoon[22], a much larger number of identities would be required for fan-out. Identity instructions are also introduced where required synchronization is not implicit in the data flow, *e.g.*, determining when an activation is complete.

The row labeled *fork&switch* includes conditional and unconditional forks in the TL0 and switch instructions in the TTDA, used to steer values into conditionals or loop bodies. The *D Loop* row reflects instructions that update tag fields and control unfolding in loops. The TL0 compilation uses 1-bounded loops[8] and would incur additional overhead with a more general parallel iteration strategy.

Compiling to threads compensates to a large measure for the additional instructions introduced for explicit scheduling in the TAM model. The increase in overall instruction counts is modest compared to more complex dataflow models. Still, if the remaining scheduling operations carry the cost of a traditional context switch, the cost of dynamic synchronization would be tremendous. The recognition of threads alone does not obviate the need for hardware support in implementing fine-grained synchronization. Rather, it is the use of a program representation that provides a very inexpensive thread switch.

3.3 Scheduling hierarchy

To demonstrate the impact of the TAM scheduling hierarchy, Table 3 presents instruction frequency and scheduling frequency measurements for the Id programs discussed above on fairly large problem sizes. This data provides preliminary estimates of thread size (I/T), threads per quantum (T/Q),

Dynamic Instruction Mix						
Run	simple 2 50	mmt 100	mmti4 100	gamteb 2048	QS 10,000	AS 500
Instructions (I)	18,000,274	25,426,150	14,476,154	9,303,069	14,767,096	8,940,066
Arith	28.6	36.5	50.8	16.1	9.1	26.7
Ifetch	10.2	7.9	14.0	6.5	4.7	7.0
Istore	1.1	0.2	0.3	2.6	2.5	0.0
Move	28.6	45.7	18.0	38.6	35.3	39.5
Fork	12.8	0.2	0.3	11.8	13.4	5.6
Jump	11.8	9.0	15.9	17.9	30.8	21.1
Rsend	5.8	0.5	0.8	5.4	2.4	0.1
Alloc	1.6	0.0	0.1	1.1	1.7	0.0
Other	0.4	0.0	0.0	0.0	0.0	0.0
% Split-phase	17.8	8.6	15.1	12.4	11.4	7.1

Dynamic Scheduling Mix						
Activations (A)	122,619	10,507	10,507	69,324	60,003	505
Quanta (Q)	351,202	21,012	21,012	179,060	159,999	1,009
Threads (T)	3,832,724	3,379,906	3,379,806	2,360,911	5,381,855	2,015,924
% Synch	44.6	32.3	32.3	43.8	31.5	25
Ave Fan-in	3.3	2.1	2.1	2.9	2.7	3.0
Q/A	2.9	2.0	2.0	2.6	2.7	2.0
T/A	31.3	321.7	321.7	34.1	94.1	3,992
I/A	146.8	2,420	1,378	134.2	246.1	17,703
T/Q	10.9	160.9	159.4	10.1	52.0	1,997
I/Q	51.3	1,210	688.9	34.1	92.3	8,860
I/T	4.7	7.6	4.3	3.9	2.7	4.4

Table 3: TL0 Instruction and Scheduling Measurements for Large Programs

and the net cost of synchronization under various degrees of hardware support.

Thread Partitioning: Although the partitioner is primitive compared to the theoretical state-of-the-art[28], thread sizes (I/T) are close to typical branch distances, which provides an upper bound given that fork is the only form of control transfer. The observed thread sizes are comparable to run lengths reported for hybrid dataflow machines, where conditionals were treated as strict[16]. Partitioning eliminates more than half of the dynamic synchronization points and, by maintaining thread-local values in registers, eliminates roughly half of the frame references and reduces the frame size substantially. There is no fundamental reason to exclude branching in the abstract machine, although with branching the definition of a thread becomes more subtle. Table 3 separates out the forks that could be transformed into simple jumps. In the absence of hardware support for multiple threads, this is a more appropriate accounting.

The non-strict semantics of the Id language place unusual constraints on thread partitioning that would be absent in less powerful languages. Excluding this factor and the absence of branching, the primary limit on the size of statically scheduled program entities is the presence of long-latency operations, represented by split-phase operations in TL0. On average, the distance between split-phase operations is between 10 and 15 instructions. Replication of data, such as might be achieved by caching, would reduce the number of these operations that actually experience long latency, but not the number that potentially do so. Thus, in any execution model that attempts to mask latency

QS(1000)	1	2	4	6	8	10	12
Defered Ifetches	3000	8973	15736	21021	23812	25006	26020
Q/A	3.0	4.0	5.3	6.3	6.9	7.3	7.8
T/Q	23.0	17.2	12.5	10.9	9.9	9.5	9.1
I/Q	63.9	47.6	36.3	30.2	27.6	26.3	25.2

Table 4: Dynamic Scheduling Measurements for 1 to 12 Processors on a Sequent Symmetry

through split-phase operations, thread-local values provide only limited opportunity for exploiting processor registers and the frequency of thread switch is non-trivial.

Quanta: Although individual threads are usually small, the number of logically related threads that can be scheduled closely together in time is often substantial. This is indicated by the row of Table 3 labeled T/Q, which gives the average number of threads scheduled per quantum. By recognizing this level in the scheduling hierarchy, it is possible to carry values in registers across threads. Observed average quantum size (I/Q) indicates that there is substantial opportunity to utilize high-speed registers at this level. The completion and initialization threads can be configured in conjunction with register allocation to save and restore registers when the compiler incorrectly predicts the quantum boundaries. Register allocation becomes challenging in this quasi-dynamic setting, as the cost metrics are probabilistic.

The thread enabled by a fork instruction is guaranteed to execute within the same quantum as the fork. Thus, the fork can be implemented by simply pushing the instruction pointer of the target thread on the local continuation vector. The thread switch (`STOP`) is simply a pop-jump. The cost of this form of thread switch is only a few instructions. The last fork in a thread can be pulled to the end of the thread and elided with the `STOP` to produce a jump or a fall-through, depending on the ordering of threads in the code-block. As indicated by Table 3, a large fraction of the forks can be transformed into jumps.[6]

The quantum figures in Table 3 must be interpreted with some caution, since this is scheduling dependent and the numbers derive from uniprocessor runs. Table 4 provides preliminary scheduling measurements for the quicksort program on multiple processors. This makes extensive use of non-strict data structures, yet the average quantum size is roughly halved on multiple processors. We are developing a fully instrumented implementation of TL0 on a 2048-node nCUBE/2 to determine quantum sizes in a truly parallel setting on large programs.

Net Synchronization Cost: Based on the scheduling data in Table 3, we can make preliminary estimates of the net cost of synchronization with or without special hardware support. Under a dataflow execution model, the primary form of synchronization is matching pairs of operands. This requires one cycle on an ETS dataflow architecture[22] or multiple cycles with a hash-based matching store[15], under the assumption that the matching store and the token queue are maintained in high-speed processor storage.[7] We see that 1.5 tokens are processed per instruction, or one synchronization event for every two instructions, on average. The synchronization cost under a dataflow model is roughly $0.5I$ times the match cost.

Since TAM employs a range of synchronization mechanisms, we must account for the possible

[6] On the other hand, the ability to separate the fork from the stop provides a generalization of delayed branch and may be advantageous to support directly.

[7] The read-modify-write process associated with the match may also stretch the machine cycle, so a small multiplicative factor should be employed to account for this and an average miss penalty should be included. Still, the basic argument is unchanged.

relationships of the participants. Suppose that A and B are two portions of the program that must complete before portion C is executed. If A and B are part of the same thread, the cost of synchronization is zero, since it is implicit in the thread ordering. With an average thread size of four, roughly half of the dataflow synchronization events ($0.25I$) fall in this category. Thus, even with extensive hardware support, compiling to threads can reduce the net synchronization cost. If A and B are in two threads in the same code block, the synchronization cost is the two entry count decrements and the fork/stop or simple jump, depending on where the fork appears in the thread. This case is represented by the instructions counted in the rows labeled Fork and Jump in Table 3. Taking the two large programs, simple and gamteb, as representative, we see 12% of the TAM instructions are subject to an overhead of roughly five machine instructions and 12% to 18% subject to an overhead of roughly three instructions. Direct hardware support for this operation can bring noticeable performance improvement, but is unlikely to provide more than a factor of two. (The quicksort example indicates a more dramatic improvement, but is also most strongly influenced by the immature state of our compiler.)

The most costly form of synchronization is the interaction of threads in different code-blocks, either through parameter and result passing or through structure operations. These are the external messages handled by inlets. The row labeled split-phase in Table 3 shows the number of these events as a percentage of the instructions executed; roughly one in eight. We may assume the event is associated with the arrival of a message, the first word of which identifies the inlet and the second word specifies the activation frame. To process such a message, the inlet is dispatched; it receives the rest of the message, stores values into slots in the specified frame, and posts a thread in the continuation vector. Eventually, the frame will be made resident and its constituent continuations processed. Thus, the cost of this form of synchronization is $net + n + a/q$, where n is the cost of the inlet, a is the cost associated with scheduling and perhaps copying state for the activation, q is the number of split-phase operations per quantum, and net is the cost of interacting with the network. Under TAM, n and a are only a few instructions and dynamic measurements indicate that q can be fairly large. Thus, by structuring the execution model in a manner that promotes large quanta, even under dynamic thread scheduling, the cost of this form of synchronization is also reduced. By making the register level of the storage hierarchy explicit and recognizing quantum boundaries, the compiler may attempt to minimize a while promoting use of registers.

The most significant synchronization cost is the actual transmission and reception of the message from the network. This has been addressed in dataflow machines in the context of elaborate hardware support for scheduling and synchronization. For example, on Monsoon a single 144-bit message packet can be transmitted or received in a cycle or two. Upon reception, it is placed on a large hardware managed scheduling queue. In essentially every commercially available multiprocessor, a combination of awkward network interface and operating system software places the cost of this operation on the order of hundreds of cycles. TAM demonstrates that efficient processor/network interaction and program level scheduling and synchronization can be addressed independently. Issuing split-phase requests is not unlike issuing instructions to a modern numeric co-processor. Receiving requests requires little more than dispatching to an inlet sequence; the message decode is "compiled in" and storage is preallocated.

4 Conclusion

In this paper, we have presented an execution model for fine-grain parallelism, in which synchronization, scheduling, and storage management is explicit and under compiler control. This is defined by a simple abstract machine with a multilevel scheduling hierarchy: instructions within a thread, threads within a quantum, quanta within an activation. This makes a range of synchronization mechanisms available to the compiler, so it can exploit frequently occuring special cases. Overall, the net cost of synchronization on conventional processors is shown to be roughly comparable to that on machines with elaborate hardware support, such as proposed dataflow architectures. An implementation of the dataflow language Id is shown to be roughly competitive with conventional languages on current uniprocessors.

Empirical measurements show that threads are small, but substantial temporal locality exists among logically related threads, even under dynamic scheduling. With scheduling made explicit, the compiler can make effective use of high-speed processor state across a sizable collection of threads, provided that it manages that resource under a quasi-dynamic scheduling paradigm. Implicit scheduling in hardware is shown to be of questionable value, as it prevents register usage beyond thread boundaries. Exposing scheduling to the compiler allows it to synthesize particular scheduling policies in specific portions of the program.

Tolerance to long, unpredictable communication latency and inexpensive synchronization in a large namespace have been put forward as fundamental architectural issues for general purpose parallel computing. In this paper, we have argued that these are primarily compilation issues. In order to generate programs that perform well on large multiprocessors, the compiler must represent the program in such a way that the most frequent forms of synchronization and context switching are extremely rapid. This can be achieved without elaborate hardware scheduling. Multiple register sets provide benefits under this execution model, but are not essential. This suggests that the fundamental architectural issues are simply: how many cycles does it take to deliver data from the user program into the network and how many cycles does it take to extract data from the network and deliver it to a useful place in the user program.

A TL0

TAM is currently represented by TL0, a 3-address instruction set with somewhat unusual control and memory operations. An overview of TL0 is provided in Table 5. Operands indicated by *src* may be frame slots, integer registers, floating-point registers, or literals. The usual integer and floating-point arithmetic and logical operations are provided in a three address form, where *dst* may be either a frame slot or a register. The operation code is qualified by type or size: `.I` for integer, `.F` for floating point, `.H` for half-word, and `.W` for full-word (64 bits).

In TL0 a thread is a sequence of instructions bracketed by `THREAD` *thr* and `END_THREAD`. Synchronizing threads start with a `SYNC` instruction which specifies a frame slot that holds the thread entry count. The `FORK` operation enables a thread for execution. The thread is specified as an immediate operand. `SWITCH` is a conditional fork; it enables one of two local threads for execution. `CASE` is a conditional fork that enables a thread obtained by indexing into a static table. `STOP` causes the current thread to be terminated and an enabled thread from the current activation to be dispatched. Each fork to a synchronizing thread causes the entry count to be decremented. The thread executes when the count reaches zero. The decrement and test may be performed at the sync or as part of the fork, in which case threads are placed on the continuation stack only if they are guaranteed to execute.

The memory operations support split-phase access to components of structures of various sizes. Thus, the read operations specify an *inlet* as a destination. The operation causes a request to be generated for the memory unit or processor containing the referenced element. The corresponding inlet places the value contained in the response into a specific frame slot and enables a specific thread. (Inlets may also process multiple values.) Three read operations are provided: `IREAD`,

Arith/Logic	*al-op.t*	*dst* = *src1 src2*
Transfer	`MOVE.`*s*	*dst* = *src*
Control	`SYNC`	*src*
	`FORK`	*thr*
	`SWITCH`	*src thr1 thr2*
	`CASE`	*src thr1 ... thrN*
	`STOP`	
Structure	`IALLOC.`*s*	*inlet* = *src*
	`IFREE`	*src*
	rd-op.s	*inlet* = *src1*[*src2*]
	wr-op.s	*src1*[*src2*] = *src3*
Call/Return	`RALLOC`	*inlet1*-*inletN* = *src1*
	`RSEND`	*src*, *inlet* = *src1.s ... srcN.s*
	`RETURN`	*res* = *src1.s ... srcN.s*
	`RELEASE`	*res* = *src1.s ... srcN.s*

Table 5: Overview of TL0 Operations

`IFETCH`, and `ITAKE`, that operate in tandem with the write operations, `IWRITE`, `ISTORE`, and `IPUT`. The operation `IREAD` simply reads the location; the split phase character is solely to avoid waiting during a long-latency request. `IFETCH` is a synchronizing form of read, that checks status bits associated with the referenced location and waits in the memory unit until an `ISTORE` fills the location. `ISTORE` causes all waiting readers to be serviced. This supports write-once data structures with element-by-element synchronization. `IPUT` and `ITAKE` provide exclusive access to a mutable location. `IPUT` stores a value into a locations and if waiters exist, a single waiter is serviced. `ITAKE` reads a full location and leaves it empty. If the slot is empty `ITAKE` waits until it is made full by a corresponding `IPUT`. Memory operations are qualified by the size of the value transferred.

The Call/Return operations support an asynchronous remote call mechanism. `RALLOC` is a split phase operation that requests a frame for a specified code-block *src1*, passing a range of inlets to be used for returning function results and inlet1 receives the pointer to the allocated frame. The frame pointer is used in `RSEND` to transfer values into an inlet of that frame. `RETURN` sends result values back to its caller. The *res* field selects which of the inlets specified in `RALLOC` to use. In addition to the operations performed by `RETURN`, the operation `RELEASE` also deallocates the frame. Typically the value returned by `RELEASE` signals completion to the parent.

Below is a simple Id function that computes the inner-product of the two input vectors `A` and `B`, over the range of subscripts 1 to n.

```
def inner_prod A B n = {
      sum = 0;
   in {for i <- 1 to n do
            next sum = sum + A[i]*B[i];
       finally sum} } ;
```

The TL0 code shown in Figure 3 begins with static declaration of frame slots, registers, and inlets, followed by the definition of the initialization thread and eight other threads. The initialization thread is executed by the RALLOC operation in the calling activation prior to any of the other threads.

```
CBLOCK inner_prod
   FRAME   LOCALS = 20, L_CVECT = 4, R_CVECT = 4
   REGS    I_REGS = 2, F_REGS = 1

# Locals:
#   L0  A,        L2  B,
#   L4  n,        L6  i,               L7  unused,
#   L8  sum,      L10 A offset,
#   L12 B offset, L14 A[i],
#   L16 B[i],     L18 thr3 synch var, L19 thr6 synch var

# Inlet declarations:
   INLET   inlet0 = thr0        # trigger
   INLET   inlet2 = L4.H thr3   # third  argument (n)
   INLET   inlet3 = L2.W thr2   # second argument (B)
   INLET   inlet4 = L0.W thr1   # first  argument (A)
   INLET   inlet5 = L10.H thr3  # A lower bound offset
   INLET   inlet6 = L12.H thr3  # B lower bound offset
   INLET   inlet7 = L14.W thr6  # A[i]
   INLET   inlet8 = L16.W thr6  # B[i]

INIT_THREAD                     # Init synch vars
   MOVE.H L18 = 4               # trigger & A & B & n
   MOVE.H L19 = 2               # A[i] & B[i]
   STOP
END_THREAD
#
# Threads receiving arguments
#
THREAD thr0                     # Trigger
   MOVE.W L8 = 0.0              # sum = 0
   MOVE.H L6 = 1                # i = 1
   FORK thr3
   STOP
END_THREAD
THREAD thr1                     # Receive first arg (A)
   IFETCH.W inlet5 = L0[0]      # fetch low bound offset
   STOP
END_THREAD
THREAD thr2                     # Receive second arg (B)
   IFETCH.W inlet6 = L2[0]      # fetch low bound offset
   STOP
END_THREAD
#
# Body threads
#
THREAD thr3                # Wait for all arguments
   SYNC L18
   FORK thr4               # enter loop
   STOP
THREAD thr4                # Top of loop - test
   MOVE.H IREG0 = L6       # load i into register
   LE.I IR1 = IREG0 L4     # IR1 = i <= n
   SWITCH IR1 thr5 thr7    # if IR1 fork 5
   STOP                    #     else fork 7
END_THREAD
THREAD thr5                    # Loop body - first half
   ADD.I IREG1 = L10 IREG0     # A low bound offset+i
   IFETCH.W inlet7 = L0[IREG1] # fetch A[i]
   ADD.I IREG1 = L12 IREG0     # B low bound offset+i
   IFETCH.W inlet8 = L2[IREG1] # fetch B[i]
   ADD.I L6 = IREG0 1          # i = i+1
   STOP
END_THREAD
THREAD thr6                # Loop body - second half
   SYNC L19                # wait for A[i] and B[i]
   MOVE.H L19 = 2          # re-init synch var
   FORK thr4               # enable loop top
   MUL.F FREG0 = L14 L16   # A[i] * B[i]
   ADD.F L8 = L8 FREG0     # sum = sum + A[i]*B[i]
   STOP
END_THREAD
THREAD thr7                # End - return result
   RETURN res1  = L8.W     # return result
   RELEASE res0            # return signal and dealloc
   STOP
END_THREAD
END_BLOCK
```

Figure 3: Sample TL0 code for inner product

Acknowledgements: This study grew, in part, out of a collective term project in EECS 294-2, that included work by Rafael Saavedra-Barrera, M. T. Raghunath, and Tzi-cker Chiueh. We have drawn heavily on the work of Arvind's Computation Structures Group at the MIT Lab for Computer Science, responsible for Id, the compiler front-end, and the basic evolution of dynamic dataflow architectures. We have profited from many helpful discussions with Greg Papadopoulos, Ken Traub, Jamey Hicks, Rishiyur Nikhil, and Brad Kuszmaul.

References

[1] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, pages 104–114, Seattle, Washington, May 1990.

[2] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. *The Int. Journal of Supercomputer Applications*, 2(3), November 1988.

[3] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[4] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proc. of DFVLR - Conf. 1987 on Par. Proc. in Science and Eng., Bonn-Bad Godesberg, W. Germany*, June 1987.

[5] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report CSG Memo 269, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, February 1987. (Also in *Proc. of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[7] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[8] D. E. Culler. Managing Parallelism and Resources in Scientific Dataflow Programs. Technical Report 446, MIT Lab for Comp. Sci., March 1990.

[9] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. of the 15th Annual Int. Symp. on Comp. Arch.*, pages 141–150, Hawaii, May 1988.

[10] W Dally and et al. Architecture of a Message-Driven Processor. In *Proc. of the 14th Annual Int. Symp. on Comp. Arch.*, pages 189–196, June 1987.

[11] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon Dataflow Processor. In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, 1989.

[12] V. G. Grafe and J. E. Hoch. The Epsilon-2 Hybrid Dataflow Architecture. In *Proc. of Compcon90*, pages 88–93, March 1990.

[13] J. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[14] R. H. Halstead, Jr. and T. Fujita. MASA: a Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. of the 15th Int. Symp. on Comp. Arch.*, pages 443–451, 1988.

[15] K. Hiraki, K. Nishida, S. Sekiguchi, and T. Shimada. Maintainence Architecture and its LSI Implementation of a Dataflow Computer with a Large Number of Processors. In *Proc. of the 1986 Int. Conf. on Par. Proc.*, pages 584–591, 1986.

[16] R. A. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proc. 15th Int. Symp. on Comp. Arch.*, pages 131–140, 1988.

[17] H. F. Jordan. Performance Measurement on HEP — A Pipelined MIMD Computer. In *Proc. of the 10th Annual Int. Symp. on Comp. Arch.*, Stockholm, Sweden, June 1983.

[18] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, pages 148–159, Sealttle, Washington, May 1990.

[20] R. S. Nikhil. Id (Version 88.0) Reference Manual. Technical Report CSG Memo 284, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, March 1988.

[21] R. S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, Jerusalem, Israel, May 1989.

[22] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, 1990.

[23] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine.* PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[24] R. Saavedra-Barrerra, D. E. Culler, and T. von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the 2nd Annual Symp. on Par. Algorithms and Arch.*, July 1990.

[25] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, pages 46–53, Jerusalem, Israel, June 1989.

[26] B. Smith. Keynote address. 17th Annual Int. Symp. on Comp. Arch., June 1990.

[27] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *Proc. of Supercomputing '88*, pages 35–41, Orlando, FL, 1988.

[28] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[29] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proc. of the 16th Int. Symp. on Comp. Arch.*, pages 273–280, Jerusalem, Israel, May 1989.