Empirical Study of a Dataflow Language on the CM-5[1]

David E. Culler
Seth Copen Goldstein
Klaus Erik Schauser
Thorsten von Eicken

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

Abstract: This paper presents empirical data on the behavior of large dataflow programs on a distributed memory multiprocessor. The programs, written in the dataflow language Id90, are compiled via a Threaded Abstract Machine (TAM) for the CM-5. TAM refines dataflow execution models by addressing critical constraints that modern parallel architectures place on the compilation of general-purpose parallel programming languages. It exposes synchronization, scheduling, and network access so that the compiler can optimize against the cost of these operations.

The data presented in this paper evaluates the TAM approach in compiling dataflow languages on stock hardware. We present data on the instruction mix, speedup, scheduling behavior, and locality of large ID90 programs. It is shown that the TAM scheduling hierarchy is able to tolerate long communication latencies, especially when some degree of I-structure locality is present. We investigate how frame allocation strategies, k-bounded loops, and I-structure caching and distribution together affect the overall efficiency. Finally we document some scheduling anomalies.

1 Introduction

The goals of dataflow models of computation are to exploit irregular or unstructured parallelism arising in general purpose programs and to enable development of high-level parallel languages in which the programmer need not manage every detail of the mapping of program and data onto the machine. It has been demonstrated that direct execution of dataflow graphs is not essential to attain these goals, as dataflow graphs are equivalent to threaded instruction sets with very weak addressing modes[9, 18]. Dataflow graphs are, however, very useful in the compilation process[4, 23, 22, 25, 28]. At the machine level, the essential aspect of dataflow is efficient dynamic scheduling. Dynamic scheduling provides tolerance to communication latency, since the processor picks up other useful work rather than waiting for each response. It avoids synchronization waits in the same manner, and further reduces the impact of transient load imbalance, since a processor remains busy as long as *any* local work is available. Dynamic scheduling supports powerful parallel languages with synchronizing data access[3] or non-strict order of evaluation[27, 7]. Dataflow processors operate greedily, grabbing hold of any available useful work, rather than sitting idle. The general belief in the dataflow research community is that if such an eager processor can be built with a reasonable cost/performance ratio, the remaining systems issues involved in actually mapping the computation and data to the machine could be solved.

Even though the research area is several years old, today we have almost no solid empirical data to substantiate this belief. There are plenty of novel ideas for implementing dynamic scheduling, but little evidence that it actually simplifies the task of managing resources or scheduling computation, or that it translates into performance on a large scale. Simulations of paper designs and small

[1]This paper was presented at the ISCA '92 Dataflow Workshop. The results are based on the 1992 version of our compiler and runtime system.

prototypes provide only limited data, since they cannot model the behavior of large programs in-the-large. The Manchester dataflow machine is correctly considered a single processor, as the multiple bit-slice-ALUs are a technological artifact. Monsoon[17] is only available in very small configurations. At the time the machine was designed, the characteristics of the per-processor storage hierarchy were not understood, so the issue was intentionally avoided by using fast static RAM for all the memory in the processor. The network operates at five times the processor clock rate, so there is little communication latency. Sigma-1 and EM-4 are available in large configurations, but the only programs that have been run are small, regular, and hand tuned. Static dataflow machines have been applied primarily to regular, structured problems[10, 11, 24]. Dataflow languages, including Sisal and a restricted form of Id, have been implemented on conventional architectures, by exploiting the regular structure in traditional FORTRAN-like applications[5, 19].

Our work attempts to fill some of this empirical vacuum by implementing Id90 on large parallel machines in a manner that retains the efficient dynamic scheduling of dataflow models. The approach we have taken is to define a threaded abstract machine (TAM) that remedies some basic shortcomings in previous dataflow models and is closer to conventional architectures. TAM exposes synchronization, scheduling, and network access so that the compiler can optimize against the cost of these operations. The Id90 compiler has been substantially rewritten to target TL0, the TAM assembly language, rather than a dataflow instruction set. A key step in the compilation process is partitioning the dataflow program graph into threads[22, 28]. The other new aspects of the compilation process involve management of registers and local storage in the context of dynamic scheduling and code generation for threads. A separate compilation step translates the TL0 code to native machine code for a variety of platforms, including large network-based multiprocessors (Thinking Machines CM-5 and the nCUBE/2), small shared-memory multiprocessors (Sequent Balance and Motorola Delta), and conventional workstations. The TLO-to-machine step focuses on specifics of the target instruction set and processor/network interface.[2] One virtue of this two-step compilation approach is that TL0 execution statistics can be collected efficiently by compiling the data collection directly into the machine code.

In this paper we provide preliminary empirical data on the behavior of large Id90 programs compiled via TAM for the CM-5. This is the first commercial machine to provide a sufficiently accessible and efficient processor/network interface allowing a meaningful study of this kind. Our active message layer on the machine[29] imposes a per-message overhead that is an order of magnitude less than that of commercial message passing systems and approaches current hardware implementations of shared-memory and message-driven models. Nevertheless, there is considerable improvement possible through hardware support, and we intend this paper to provide grist for the design of the next generation of machines supporting dataflow languages, *e.g.*, *T and EM-5.

The paper is organized as follows. Section 2 explains the TAM model and briefly outlines how Id90 programs are compiled to TAM and then to the CM-5. Section 3 provides crude program performance data, such as speedup and program behavior, including TL0 level instruction mixes, scheduling behavior, and locality. Section 4 discusses the effects of resource management policies including the effects of a variety of frame allocation policies, efforts to reduce contention, including I-structure spreading and caching and the impact of k-bounded loops. Section 5 documents some disturbing scheduling anomalies.

Briefly, the main observations of the study are as follows. It is possible to implement a dataflow language on stock hardware and provide fast dynamic scheduling, although current processor network interfaces are inadequate. The TAM scheduling hierarchy appears to work even under significant latency, especially when some degree of I-structure locality is obtained. Simple frame allocation policies appear to do a reasonable job of balancing the computational load while pre-

[2]A generic back-end translates up to C and uses the local C compiler as an assembler.

serving a significant degree of locality. However, I-structure load can be extremely unbalanced, resulting in significant contention. Dynamic scheduling of work on a per-processor basis does little to mediate the effects of persistent load imbalance or contention. This must be be addressed in the way work and data are assigned to processors. Replication of I-structure data can significantly reduce contention, at a cost. Finally, in a dataflow implementation a host of factors interact in complex ways: scheduling, assignment of work, allocation of I-structures, and I-structure reference patterns. The absence of a high-level program execution strategy makes these interactions difficult to understand or control.

2 Threaded Abstract Machine

In this section, we describe TAM[8], a threaded abstract machine that serves as an intermediate step in compiling the dataflow language Id90 for conventional parallel (and sequential) architectures. Historically, Id90 was developed in close connection with dynamic dataflow architectures, especially the MIT Tagged Token Dataflow Architecture. It demands the dynamic scheduling and tagged heap storage that these designs offer. However, the extent to which these capabilities need to be supported directly in hardware remains an open question. The evolution of the MIT dataflow architectures has been driven primarily by advancement in compiler technology; each step in understanding how to compile the language resulted in a simplification of the the architecture. TAM represents an effort to simplify the architecture even further, relying heavily on sophisticated compilation techniques. Traub's "compilation as partitioning" framework[26] and Iannucci's thread generation for the hybrid architecture[14] demonstrated that it was possible to reduce the amount of dynamic scheduling required. TAM builds directly on this work, but it addresses three other issues as well. First, there is no hardware management of storage resources. More precisely, there is no implicit storage allocation in the machine model.[3] Storage is explicitly allocated in large chunks, namely activation frames and heap data structures, and the compiler is responsible for storage management. Second, the low-level scheduling of computation is focused to enhance the locality of reference within each processor, and allows efficient dynamic scheduling on conventional processors. Third, the high level scheduling of computation is exposed so that a global strategy can be "compiled in" the program. (Our TAM implementation provides this level of control to the Id90 compiler, but we have not fully exercised it.) These goals are quite compatible and can be addressed within a simple run-time program structure, described below.

2.1 Activation frames

The key to understanding TAM and its relationship to dataflow models is to examine the requirements of a function invocation. In a conventional sequential language, a function is invoked by allocating storage for local variables on the stack (the activation frame), pushing arguments onto the stack, and transferring control to the entry point of the function. The key difference in a parallel language is that the caller does not suspend on every invocation, so it may invoke many other functions to run concurrently. Thus, the dynamic call structure at any point in time forms a tree, rather than a stack, which grows and shrinks over time. Dynamic dataflow architectures implicitly allocate storage for the invocation tree, since the matching store allocates storage on a token-by-token basis. However, if we examine the language implementations on the TTDA[1] or Manchester machine[13] more carefully, we see that the function invocation involves allocating a "context" or portion of the tag space. In effect, this allocates an entire region of addresses to the

[3]Our current implementation of deferred reads backs off from this requirement slightly for performance reasons, but this deviation is not fundamental.

function invocation; the matching store is simply a means of representing a sparsely populated address space. Of course, the other novelty in the dataflow approach is that each argument transfers control to an entry point of the function. Thus, we may think of each argument as initiating a thread of control within the function invocation. Threads are implicitly synchronized and forked in the dataflow graph representation.

The Explicit Token Store[9] model returns to the conventional idea of allocating storage for local variables with each invocation. It makes the further assumption that the entire invocation will execute on the processor to which the activation frame belongs. Monsoon[17] associates a presence-bit with each frame slot to support the dataflow view which associates a thread of control with every element of local data. Hybrid[14] adopted a complementary view, providing explicit threads of control which suspend upon access to a frame slot that is marked not-present. P-Risc[16] observed that presence-bits can be kept in the frame like local data, rather than as special tags, and that matching could be simulated by toggling the tag bit atomically and suspending on the result.

What none of these models provide is a way of referring to the set threads associated with a particular function invocation. They all rely on a scheduling queue that is outside the run-time program structure; this is the token queue in dataflow machines and Monsoon. As a result, there is no way of articulating what executes next after a thread stops or suspends. Hence, the locus of computation on each processor hops around arbitrarily from thread to thread. This prevents effective use of a modern processor storage hierarchy, including a large processor register set, sRAM cache, and dRAM main store.[4] Moreover, this queue may grow arbitrarily large (as can the activation tree), since it must represent all the parallel threads in the program. The novel contribution in TAM is to maintain the scheduling queue within the collection of activation frames, as suggested by Figure 1. A portion of each frame is used to hold a stack of instructions pointers, called the *continuation vector* (CV), representing the enabled threads for the corresponding function invocation. The compiler can determine the maximum size of this region in a manner much like register allocation. The scheduling queue on each processor is formed by simply linking together frames containing enabled threads.[5] This simple scheme addresses our goals, as follows:

- The scheduling queue is no longer a specialized resource. There is a single resource that can grow arbitrarily, the activation tree, and it may occupy all of memory, just like the stack in a conventional language.[6] It is allocated only upon function invocation.

- The TAM representation dictates a natural scheduling hierarchy which enhances the locality of reference within the processor. When a frame is scheduled, threads are executed from that frame until none remain in the CV. We call this dynamic "chunk" of work a *quantum.* Focusing on the work for a frame should improve the effectiveness of the processor cache. More importantly, the processor registers are valid across threads within a quantum. We expect that in practice quanta will include several points of potential suspension, since often multiple arguments to a function arrive close together in time or multiple remote fetches will complete together.

- Dynamic scheduling among threads within the same function is extremely inexpensive. To fork a new thread, the address of the thread is simply pushed onto the CV. When a thread stops, the next thread is popped off the CV. Coordination among threads is implemented

[4]This was precisely why the initial design of Monsoon relied on a large static RAM, rather than more cost-effective cache structure.

[5]Observe that the instruction pointer in the frame is a full continuation, since the frame pointer portion is determined upon traversing the link to the frame.

[6]As with conventional languages, there is also a heap for arrays and non-local data.