Dataflow Execution of Sequential Imperative Programs on Multicore Architectures

Gagan Gupta and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI, USA
{gagang, sohi}@cs.wisc.edu

ABSTRACT

As multicore processors become the default, researchers are aggressively looking for program execution models that make it easier to use the available resources. Multithreaded programming models that rely on statically-parallel programs have gained prevalence. Most of the existing research is directed at adapting and enhancing such models, alleviating their drawbacks, and simplifying their usage. This paper takes a different approach and proposes a novel execution model to achieve parallel execution of **statically-sequential** programs. It dynamically parallelizes the execution of suitably-written sequential programs, in a dataflow fashion, on multiple processing cores. Significantly, the execution is race-free and determinate. Thus the model eases program development and yet exploits available parallelism.

This paper describes the implementation of a software runtime library that implements the proposed execution model on existing commercial multicore machines. We present results from experiments running benchmark programs, using both the proposed technique as well as traditional parallel programming, on three different systems. We find that in addition to easing the development of the benchmarks, the approach is resource-efficient and achieves performance similar to the traditional approach, using stock compilers, operating systems and hardware, despite the overheads of an all-software implementation of the model.

Categories and Subject Descriptors

C.1.3 [**Processor Architecture**]: Other Architecture Styles – *data-flow architectures;* C.1.4 [**Processor Architecture**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming*.

General Terms: Performance, Design.

Keywords: Dataflow, multicore, programming, determinacy.

1. INTRODUCTION

As parallel computers are becoming commonplace, the computing community has been aggressively seeking execution models that will permit parallel program execution. Much of the research community's efforts to date have tried to adapt the "canonical" parallel computing principles that were developed for high-end scientific computers, and then to address the myriad of issues that arise in doing so.

Briefly, a canonical parallel processing model requires the programmer (or some software tool) to identify the parallelism in the program and *statically* create a *parallel program* using a programming model such as Pthreads, MPI, or task programming, expressed in an imperative programming language such as C or C++. When such a *statically-parallel program* is executed, it results in a *parallel execution* of the program's operations. During dynamic execution of the statically-parallel program, a multitude of complexities may arise that make program development onerous [18]. For example, avoiding data races to ensure correct execution may expose users to the intricacies of the underlying architecture, such as memory consistency; unknown or improper locking protocols can cause deadlocks; complex algorithm structures may be needed to exploit available parallelism; non-deterministic execution can lead to undetected bugs, and on occasion requires recreation of the original sequence for operations such as I/O. Concurrent program analysis to help alleviate some of these issues is provably undecidable, ultimately burdening the user to reason about their correctness [24]. Yet, such a canonical model is at the core of almost all (if not all) of the parallel programming/execution models that have been proposed and practically deployed. Despite significant research efforts over the past several decades to overcome the shortcomings of the model, a practical paradigm for the future remains elusive.

An alternate model for parallel processing, the *dataflow model*, has been demonstrated, in academic research settings, to be very successful at unlocking parallelism in a class of applications. However, its adoption has been hampered by reliance on functional languages which are hard or inefficient to use for many practical applications, as well as the need to start from scratch with all (software and hardware) components of computing. A recent paper by Denning and Dennis [9] brings forth many of the issues related to the canonical parallel processing model and dataflow computing.

As we search for a practical execution model for future parallel computers, it is instructive to study the history of instruction-level parallel (ILP) processors. Here there were two major proposals: VLIW and dynamically-scheduled superscalar [20]. The former required a statically-created ILP program, whereas the latter achieved ILP by dynamically carrying out an instruction-level dataflow execution of a sequential program. The two approaches had some similarities, e.g., how functional units may be

MICRO'11, December 3–7, 2011, Porto Alegre, Brazil.

organized, but there were many differences. The former required additional architectural support (e.g., predicated execution), had much more sophisticated software (e.g., trace and hyperblock scheduling), required support to identify problems that occurred in different places than they would have occurred sequentially (e.g., support for traps), and was limited in its ability to exploit ILP when dependence information was not known until run time (e.g., dynamically-linked libraries). The latter, devoid of such problems, was able to extract and exploit dynamic parallelism, but required additional hardware logic, and potentially suffered when parallelism was abundant and was easily identified statically.

The canonical parallel execution model is the VLIW analog in the multicore world: static representation, much more sophisticated compilers, architectural support for efficient synchronization primitives, problems such as race conditions and non-determinism that do not occur in a sequential program, limited ability to achieve parallel execution when dependences are not known statically, etc. The obvious question then is: can we have a "superscalar" analog in the multicore world, i.e., can we achieve a dataflow parallel execution of a statically-sequential program, written in a ubiquitous imperative programming language, such as C++, on multicore architectures?

In this paper we answer the above question by proposing an execution model that determines data dependences between computations dynamically and executes them concurrently in dataflow fashion (§ 2). Importantly, we retain the conventional sequential imperative programming interface and our current implementation of the model requires no changes to existing compilers and operating systems, and runs on stock hardware. We describe the programming interface and the working of the model in detail (§ 3). Evaluation (§ 4) shows that it: (i) achieves performance comparable to traditional parallel programming techniques, (ii) eases programming, (iii) while incurring reasonable overheads. Finally, we compare our work with other related work (§ 5) before concluding (§ 6).

2. DATAFLOW EXECUTION OF SEQUENTIAL IMPERATIVE PROGRAMS

The dataflow model performs data-driven execution of programs [3]. Instead of executing instructions sequentially as per the control flow, it executes them as soon as their input operands are available, execution resources permitting. Dependent instructions are automatically serialized while independent instructions may execute in parallel. The model naturally exposes all of the innate parallelism within a program, while ensuring determinate execution, and remains an idealized standard for exploiting concurrency. To successfully employ a similar model in the multicore realm, we address some of the inherent and new challenges it poses: a practical programming paradigm, dependence and resource management, and application of the principles at a multicore scale.

Programming

Proposed dataflow machines have traditionally relied on functional languages to express programs. Functional languages disallow side effects and mutable data, properties that lend well to the dataflow model. But they have failed to attract the programming community. Hence we adopt the more familiar and established imperative programming language, such as C++. Developers today follow modern programming principles and practices, which encompass modularity, object oriented designs, data encapsulation, information hiding, and well defined module interfaces [6]. In our model we exploit such common-case empirical behaviors while provisioning for the rare worst-case scenarios. Further, we favor the well understood notion of a statically-sequential program over the statically-parallel multithreaded model, the reason for which we highlight later in this section.

Dataflow on Multicores

While traditional dataflow machines exploit instruction level parallelism (ILP), we raise the granularity to functions which are already used to organize task-level computations, facilitate code reuse and compose well engineered programs. More appropriate for the scale of multicores, we thus seek to exploit *Function-Level Parallelism (FLP)* by executing functions on the cores in a dataflow fashion. Employing FLP has the added potential advantage of exploiting locality, which was difficult to achieve in the dataflow machines[1].

At run time, we sequence through a sequential program, a function (rather than an instruction) at a time. Before executing the function we first need to ascertain its "operands". In our model we also raise the granularity of an operand from an individual register or memory location to an *object*. Commonly used in modern programming practices, an object may comprise of fields and usually forms a basic unit of data. A function's input operands, i.e., the *read set*, and its output operands, i.e., the *write set*, collectively called the *data set*, are readily available from its interface. Often objects in the data set may be unknown statically. Therefore we evaluate the data set dynamically, at run time, just before the function is invoked.

We use the identity of the objects in the data set to establish the data dependence between functions, in contrast to establishing independence as is done in the statically-parallel model. In particular, we determine if the function currently being processed is dependent on any prior function(s) that are still executing. If not, it is *submitted* (or *delegated*) to a core for execution. If so, it is *shelved* until its dependences are resolved. In either case, we then proceed to process the next function in the sequential program.

Handling Dependences

Dataflow machines handle dependences using *tokens* to signal production and availability of data. We employ a similar technique, but make two crucial enhancements. First we associate tokens with objects instead of individual memory locations, to match the data abstraction. Second, we assign each object multiple *read tokens* **and** a single *write token*, to manage both production and consumption of data. This is needed to handle the ability of imperative languages to mutate data as we illustrate below.

When the execution encounters a function to be considered for dataflow execution, it requests read (write) tokens for objects in the function's read (write) set; it is ready for execution only after it has acquired all its tokens. Upon completion, it relinquishes the tokens which are then passed to the shelved function(s), if need be. When a shelved function has acquired its requisite tokens, it can be *unshelved* and submitted for execution.

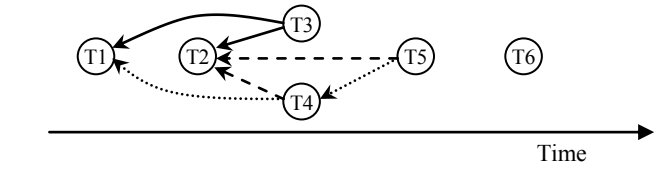
[1] We do not address this further in this paper.

```
1  while (cond) {
2    ..
3    function T: {wr_set} {rd_set}
4  }
5  function T': {?} {?}
```

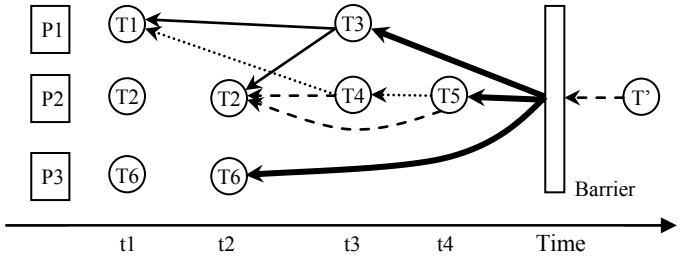
(a)

```
1    T1: {B, C} {A}
2    ..
3    T2: {D} {A}
4    ..
5    T3: {A, E} {F}
6    ..
7    T4: {B} {D}
8    ..
9    T5: {B} {D}
10   ..
11   T6: {G} {H}
12   T': {?} {?}
```

(b)



(c)



(d)

Figure 1. (a) Example pseudocode that invokes functions T and T'. *T: {write set} {read set}* modifies (reads) objects in its *write set* (*read set*). Data set of T' is unknown. (b) Dynamic invocations of the functions *T* and *T'*, in the program order, and the data set of each invocation. (c) Dataflow graph of the dynamic function stream. (d) Dataflow execution schedule of the function stream.

Imperative programming can impose additional hurdles. A "poorly" composed function (e.g., one with side effects) or an unknown function (e.g., from third party) may make it difficult to ascertain the read and write sets. In such a case our model can resort to *sequential execution* of such a function, i.e., all prior computations in the program order complete before performing the function, and any subsequent computation begins only after its completion. This precludes the need to determine precise data dependences for the function, and hence its data set.

Model Overview

We illustrate the execution model with the simple sequential program example of Figure 1a, which invokes a function T within a loop. Figure 1b shows an example dynamic sequence of invocations of T, along with their dynamically computed write and read sets, generated by sequencing through the program. Figure 1c shows the data dependence between the functions. For example, T3 writes objects A and E, and thus it has a WAR dependence (solid arrows in the figure) on T1 and T2, which read A. Likewise T4 has a RAW dependence (dashed arrows) on T2, and T5 has a WAW (dotted arrows) dependence on T4. These dependences must be preserved if the dynamic execution is to maintain the sequential appearance of the static program.

Figure 1d shows the execution of the code as per our model. The execution first encounters invocation T1. Attempts to acquire a read token for object A, and write tokens for B and C are successful. Hence T1 is submitted for execution on an available core (P1). The execution proceeds (on a processor not shown) and processes T2 while T1 is executing. Attempts to acquire a write token for D and a read token for A are successful, and thus T2 is scheduled to execute (on core P2). The execution advances to T3, which is shelved because a write token for object A can't be acquired since T1 and T2, which are still being executed, hold read tokens for A. However, before being shelved, T3 acquires a read token for F and a write token for E. Next, functions T4, T5, and T6 are processed in turn. T4 and T5 are shelved because they require a write token to B, which is being held by T1. On the other hand, T6 is able to acquire its requisite tokens (for G and H) and thus can be executed, possibly even in parallel (on core P3) with T1 and T2.

When T1 completes its execution at time t2, it releases the write tokens for B and C (signaling availability of B and C). The token for B is passed to the shelved function T4, since it is the oldest function waiting for it; T5 continues to wait until T4 finishes and releases the token for B. The token for C is returned to the object since no function is waiting for it. At this point, none of the shelved functions (T3, T4, or T5) can be woken up since none of them has successfully acquired all its tokens. When T2 completes, it releases the tokens for D and A (signaling availability of D and completion of use of A), thus waking up T3 (awaiting write token to A) and T4 (awaiting read token to D). These functions can thus be executed starting at time t3 (e.g., on cores P1 and P2), with T3 now being able to write to A in the same order as in the sequential program (after T2 is done reading it). When T4 completes, it passes the write token for B to T5, which can then be executed at time t4.

Note that dependences, even between dynamic instances of the same function, may or may not manifest at run time, e.g., dynamic instances of the function T, T1 and T2 are independent while T2 and T3 are not. Handling the dependences statically, as is required in the statically-parallel model, often leads to an overly conservative solution. By detecting and only serializing the dependent functions dynamically, while permitting independent functions to proceed in parallel we can achieve the ideal dataflow schedule of execution (resources permitting).

In the event a function's data set cannot be determined, e.g., for function T' in Figure 1a, we resort to its sequential execution, in which an implicit barrier is created between T' and all previously scheduled functions, as shown by dark solid arrows in Figure 1d. T' will be submitted for execution after all previous functions complete. The program subsequent to T' may revert back to parallel execution after T' has finished.

Deadlock Avoidance
The token mechanism in a normal dataflow model could deadlock if two or more functions create a cyclic dependency on tokens. For example, invocations T4 and T5 may create a request sequence T4:B (acquired) → T5:B (waiting) → T5:D (acquired) → T4:D (waiting) and deadlock. We avoid token deadlocks by ensuring: (i) token requests are processed one function at a time, and (ii) tokens for an object are granted in the order they are requested. Thus T5's token requests are only processed after T4's, and T5 can only receive tokens to its objects after all previous requesters have relinquished them.

Resource Management
Dataflow machines can easily scout an entire program for parallelism even when only a fraction of the program has actually executed. In the process they can exhaust resources, causing deadlocks [8]. We prevent such deadlocks by unraveling the (sequential) program only as much as the resources permit and at the same time guarantee forward progress since functions already processed are always independent of their successors.

Benefits of Sequential Programming
The sequential programming model affords the key property of an implicit order for computations, the benefits of adopting which we highlight here. First, since the computation schedule is based on the program order the model achieves sequentially determinate execution [17]. Sequential determinacy ensures that in any execution of a program with the same inputs, an object is assigned the same sequence of values. This makes programs easy to reason about, and their execution predictable and repeatable, key distinctions from the multithreaded model. Second, sequential unfolding of execution helps avoid the two types of deadlocks. Third, sequential operations such as I/O require no special handling from the user. Finally, it obviates the need to abandon what is already a very well established and understood programming paradigm.

Just as it was in the case of dynamically scheduled superscalar processors, we believe that the proposed model is an apt execution model for parallel processing hardware and thus will significantly impact future hardware and software architectures. But before we can investigate how to architect the hardware and software components to support the model, we first study its feasibility. Instead of building a simulator of the hardware components, we chose to build a prototype entirely in software to experiment on real hardware. Such an approach has the potential to incur high overheads, but rapidly allows us to assess the practicality and viability of the approach. The remainder of the paper describes and evaluates the prototype. As we show in the evaluation section, despite the overheads of a software implementation it achieves performance that is competitive with traditional parallel execution models on commodity hardware.

3. PROTOTYPE IMPLEMENTATION
We developed a software prototype of the execution model described in section 2 in the form of a C++ runtime library. The application to be parallelized is compiled with the library which becomes a part of the program that it dynamically parallelizes. During execution the runtime provides three key capabilities: (i) identify and track dynamic dependences, (ii) schedule functions for execution while balancing load, and (iii) orchestrate dataflow parallel execution of independent computations. As part of the executable of the user-level code, it executes on the same hardware and uses the same system memory as the application. We next describe program development using the library, and the runtime mechanics.

```
1  while (cond) {
2    ..
3    df_execute(wr_set, rd_set, &T);
4  }
5  df_seq (G, &print);
6  df_end ();
7  T' (args);
```

Figure 2. Example program in the proposed model.

3.1 Static Sequential Program
To enable full expression of imperative languages, the model permits programs to express and alternate between dataflow and sequential execution. Programs written for the model, currently in C++, are similar to conventional sequential programs, augmented to work with the prototype. Specifically, in addition to coding the algorithms, users identify: (i) potentially parallel functions therein, (ii) objects shared between such functions, (iii) their read and write sets, and (iv) sequential program segments, if any.

Dataflow Functions
A program written for the model may be viewed as comprising of a *main program context* from which functions **may** be invoked for concurrent execution. A program function to be considered for potential parallel execution is invoked using the *df_execute* interface provided in the library. *df_execute* is a runtime function implemented using C++ templates. The function pointer and its arguments are included in the call to *df_execute*. It prompts the runtime to attempt execution of the function in parallel with the *continuation* of the program, i.e., the remainder of the program past the *df_execute* call, and other (sequentially preceding) executing functions, dependences permitting, as we shall see below. Non-*df_execute* instructions in the program execute in the specified sequence. Figure 2 shows the main program context of an example similar to the one in Figure 1a. Function T is invoked on line 3 by passing its pointer to *df_execute*, possibly executing it with other executing functions and the continuation beginning from line 4.

Users also identify object classes that may be shared across parallel functions by inheriting from a token base class supplied in the library (not shown in the example). The runtime associates tokens with the dynamic instances of these classes; they are used to track dependences as we describe in the following subsections.

Shared data, in the form of globals, passed-by-reference objects or pointers to them, that are accessed by a function are passed to it as arguments. Users group them into two sets, one that may be modified (write set) and another that is only read (read set). In case the users are unable to create precise sets, they may specify a superset of objects. The C++ STL-based *set* data structure of the token base class is used to create them. The two sets are also passed to the function via *df_execute*, as shown in Figure 2 on line 3 for function T (wr_set and rd_set).

Users may also pass data not shared across parallel functions and passed-by-value objects through *df_execute* to the function, by listing them after the function pointer (none in the example). Tokens need not be associated with such data since they do not cause dependences and are inconsequential to parallel execution.

The user formulates the read and write sets before invoking *df_execute*. The runtime determines the identity of the objects in

them, by dereferencing pointers if need be, at run time. This interface allows the model to handle data referenced using pointers, as well as pointer arithmetic.

Serial Segments/Functions

Users may revert to sequential execution of program segments by using the *df_end* interface, as shown on line 6 in Figure 2. *df_end*, analogous to a barrier, causes the runtime to suspend execution of the main program context and quiesce the dataflow execution. The program resumes after all previous functions finish, thus serializing execution of the computation that follows *df_end*, T' in our example. Beginning from function T', the program receives no special treatment from the runtime until the next *df_execute*.

To operate on a shared object in sequence with the main program, the runtime provides a *df_seq* interface. *df_seq* accepts the object instance, the function (object method) pointer and any arguments to it. *df_seq* achieves *object-sequential execution*, which we define as one in which all prior computations that access the specified object complete before the given computation begins, and any subsequent computation in the program order can only begin after it completes. *df_seq* causes the runtime to suspend the main program context until the associated function finishes operating on the specified object. Line 5 in Figure 2 shows use of *df_seq* to invoke the function *print* on shared object G. Execution will proceed from line 6 only after *print* finishes, but potentially in parallel with other (prior) functions (that are not accessing G).

While *df_end* permits the user to quiesce the dataflow execution with respect to all objects and safely fall back to sequential execution where parallel execution may be undesired or difficult to use, *df_seq* allows the user to quiesce the parallel execution with respect to a single object.

Beyond inheriting from the base class for the shared objects, creating the read and write sets, and invoking the potentially parallel functions via *df_execute*, the onus is **not** on the programmer to ensure mutual exclusivity between the functions or insert and manage any synchronization. While additional to conventional sequential programming, we believe this imposes only a modest burden on users who already follow many modern software engineering principles, and much less than reasoning about correctness of and debugging non-deterministic statically-parallel programs.

3.2 Runtime Mechanics

Our runtime, based on the Prometheus runtime described by Allen [2], employs multithreading to implement the mechanics of parallel execution using the Pthreads API. However, the thread management is abstracted away from the user. In fact, the user is entirely oblivious of the underlying infrastructure and its mechanics, such as the number of hardware contexts, atomic operations, synchronization activities, etc.

Executing Functions on Processing Cores

At the start of a program, the runtime creates threads, usually one per hardware context available to it. A double-ended work queue (deque) is then assigned to each thread in the system. Computations are scheduled for execution by a thread by queuing them in the corresponding work deque. Each thread also has a task-stealing scheduler whose use will be seen below.

Discovering Functions for Parallel Execution

At the onset, all but one of the processors idle, waiting for work to arrive in the deques. Execution of the program begins on a single processor core and unfolds in a fashion similar to sequential

```
1  Prelude () 1 {
2    Acquire tokens 1.1 {
3      // See Figure 4b
4    }
5    If all tokens not acquired {
6      Shelve function 1.2
7    }
8 }
9 // All tokens acquired
10 Execute function () 2
11 // Return from function
12 Postlude () 3 {
13   Release Tokens 3.1 {
14     // See Figure 4c
15   }
16 }
```

Figure 3. Logical view of runtime operations to process a dataflow function.

execution. When a function to be considered for parallel execution (via *df_execute*) is encountered, the runtime is activated. The runtime processes a dataflow function in three decoupled phases, *prelude*, *execute* and *postlude*, as outlined in Figure 3. In the prelude phase (Figure 3: 1) it dereferences pointers to objects in the read/write sets, if need be, and attempts to acquire the tokens. Successful acquisition of tokens leads to the execute phase (Figure 3: 2), in which the function is *delegated* for (potentially parallel) execution. Specifically, the runtime pushes the program continuation (remainder of the program past the *df_execute* call) onto the thread's work deque, and executes the function on the same thread. A task-stealing scheduler, running on each hardware context, will cause an idle processor to steal the program continuation and continue its execution, until it encounters the next *df_execute*, repeating the process of delegation and pushing of the program continuation onto its work deque. Thus the execution of the program unravels in parallel with executing functions, and possibly on different hardware contexts rather than on one hardware context. The postlude phase is described later in the subsection.

Tokens and Dependency Tracking

During the program execution, when a designated shared object is allocated it receives one write token, unlimited read tokens (limited only by the number of bits used to represent tokens), and a wait list. Tokens are acquired for objects that the dataflow functions operate on, and released when the functions complete. A token may be granted only if it is *available*. Figure 4a gives the definition of availability of read and write tokens, and Figure 4b shows the token acquisition protocol. The wait list is used to track functions to which the token could not be granted at the time of their requests. A non-empty wait list signifies pending requests, in the enlisted order. To prevent deadlocks all token requests from a function are processed before proceeding to the next function, possibly in parallel with other executing functions, and a token is granted strictly in the order in which it was requested. Hence an available token is not granted if an earlier function enqueued in the wait list is waiting to acquire it (Figure 4b: 1). For example, if a read token is available but a prior function requires a write token to the object, it cannot be granted to the requester, ensuring that functions acquire tokens in the program order.

Shelving Functions/Program Continuations

If the tokens for a function cannot be acquired, it is enqueued in the wait lists of all the objects for which tokens could not be acquired (Figure 4b, 4 or 5), and subsequently shelved (Figure 3: 1.2). While the shelved function waits for the dependences to

```
Initial State:
  Read Token:  None granted.
  Write Token: Not granted.

Read Token is available if:
  write token is not granted.

Write Token is available if:
  no read token is granted AND
  write token is not granted
```

(a)

```
Acquire Read/Write Token:

1 if wait list is empty { 1
2   if token is available { 2
3        grant token 3
4   } else {
5     enqueue in wait list 4
6   }
7 } else {
8   enqueue in wait list 5
9 }
```

(b)

```
Release Token (Read/Write):

1 return token 1
2 while wait list is not empty { 2
3   if token is available {
4     grant token
5     if waiting function is ready {
6       schedule function 3
7     }
8    } else {
9      break
10 }
```

(c)

Figure 4. The token protocol: (a) Definition of availability, (b) Read/Write token acquisition, (c) Token release.

resolve, the runtime looks for other independent work from the program continuation to perform, as above.

df_seq can cause the program continuation to be shelved (possibly in addition to the associated function). The continuation is enqueued in the wait list of the object identified in *df_seq*. *df_end* can also cause the program continuation to be shelved until all executing functions complete. In this event the continuation is shelved on a special runtime structure. Since the program is now suspended in either case, the context looks for previously shelved functions that may now be unshelved, via task stealing.

Completion of Function Execution

When a dataflow function completes execution it returns the control to the runtime, which initiates the postlude phase (Figure 3: 3). In this step all acquired tokens are released. Figure 4c describes the token release protocol. Once a token for an object is relinquished (Figure 4c: 1), it is passed to the next waiting function(s) in the wait list (Figure 4c: 2), if present, in the same order that the functions were enqueued. When a shelved function receives all requested tokens, it is considered to be *ready* for execution. The thread that grants the final token also awakens the ready function and schedules it for execution by enqueuing it in its own work deque (Figure 4c: 3). Then either the same thread, or another that steals the function, will eventually execute it (Figure 3: 2). The process is identical if a *df_seq* caused the program continuation to be enqueued in a wait list. Different threads may simultaneously attempt to release, acquire, and pass a token. The runtime manages the data races in such scenarios.

The runtime tracks in-flight functions by updating a count every time a function is processed or completed. If *df_end* caused the continuation to be shelved, the postlude phase of the last function to finish will schedule the continuation for execution, resuming the program.

Thus, by shelving dependent functions, scheduling them for execution as soon as their dependences have resolved, and executing other independent functions in the meantime, the runtime achieves the dataflow execution outlined in our model.

Scheduling and Balancing Load

To achieve optimal performance we employ dynamic task-scheduling similar to the one outlined in Cilk-5 [12] and [2]. The runtime uses the work-first principle and lazy task creation to delegate functions to threads, and a randomized task-stealing policy to balance the load. Threads first execute functions from their own deques when available, failing which they steal from others. We make three enhancements to the algorithm: (i) we employ polymorphic work deques that can hold functions and program continuation, (ii) program continuation and functions may be shelved, and (iii) the thread that awakens a shelved function enqueues it in its own deque, potentially executing it next. Doing so permits the dependent function to migrate to the thread that may have already cached the data. Furthermore, all threads have a runtime scheduler, hence we avoid bottlenecks that result from using centralized resources.

3.3 Example Execution

We illustrate the runtime operations using the example code in Figure 2. Figure 5a shows the initial state of the system. CPU0 is executing the program and objects have been constructed. Objects A to F are depicted along with their tokens. R = *n* indicates *n* read tokens have been granted; W indicates the write token has been granted (none at this time). Wait lists of objects A, B and D, currently empty, are also shown. Figure 5b shows the state of the system as the program begins execution (the same while loop as in Figure 1a). Events are identified using # time stamps. The first invocation of T, T1, with write set {B, C} and read set {A} is encountered. Write tokens for B and C, and a read token for A are acquired (1) and the function is delegated for execution on CPU0 (1), and the program continuation is pushed on to the deque of CPU0. From there, CPU1 steals the continuation, and encounters the second invocation, T2. It acquires tokens for objects D and A, and delegates T2's execution on itself (2). Note that at this time, two read tokens for object A have been granted.

Next, the third invocation, T3, attempts to acquire a write token for A, and fails (however it succeeds in acquiring tokens E and F (3)). It is hence shelved and enqueued in the wait list of A (3). The next invocation, T4, also fails to acquire a read token for object D and a write token for B, and is enqueued in their wait lists (4). Likewise, the following invocation, T5, has to be enqueued in object B and D's waitlists (5). While the shelved methods await their tokens, the main program context on CPU2 reaches the final invocation of T from the loop, T6, which is delegated for execution on CPU2 itself (6).

Upon completion T1 releases the tokens back to objects B, C and A (7 in Figure 6a). This causes B's write token to be granted to T4 (8). However T4 cannot execute yet since not all of its requested tokens have been granted. Next CPU0 steals the program continuation from CPU2 and encounters *df_seq*. It causes the runtime to shelve the program continuation beyond *df_seq* in G's wait list (not shown) and await completion of all functions accessing G (only T6 in this case), before advancing further.

Eventually, T2 and T6 complete (Figure 6b), and release their tokens (9). Now that all of its read tokens are available, the write token for A is granted to T3 (10). Since T3 now has all its tokens, it is enqueued in the work deque, from where the next available context will execute it (10). Similarly, once the write token is returned to object D, read tokens are granted to T4 and T5 (11,

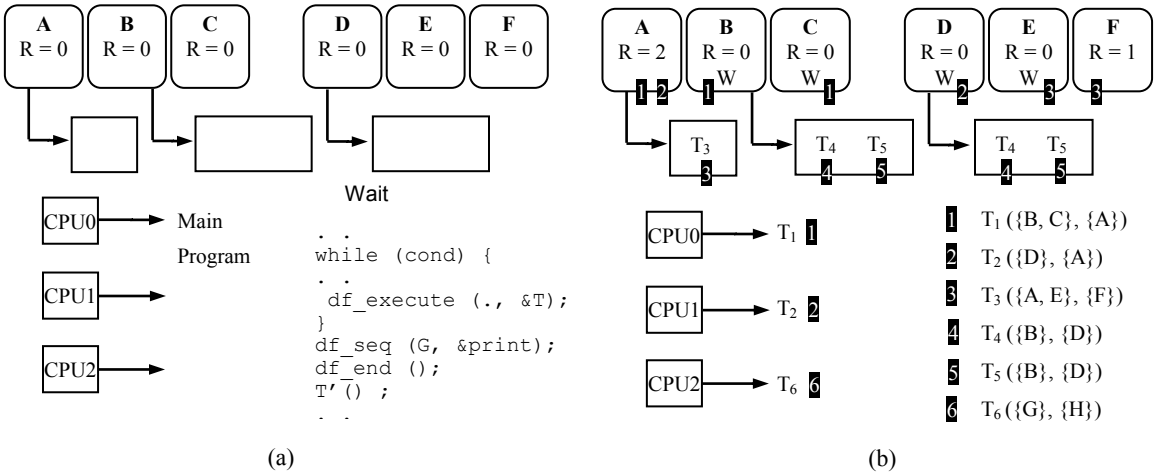


Figure 5. Example execution. (a) Initial state of the system. (b) State of the system after processing the six invocations of function T, denoted as: T {write set} {read set}.

12). This causes T4 to transition to the ready state, and be scheduled for execution on an available context (11). Completion of T6 causes the runtime to execute the *print* method on object G (13). T5 will be scheduled for execution once T4 completes and releases the tokens for objects B and D. After *print* completes, the runtime schedules the program continuation for execution, whereupon it processes *df_end*. The continuation is shelved again, thus preventing further processing of the program, until all in-flight functions (only T5 here) finish, before proceeding to T'.

4. EVALUATION

We evaluated the software prototype of the proposed model for its programming efficacy and performance efficiency by developing several benchmarks using the runtime library. Table 1 lists the applications we selected from different benchmark suites and the input data sizes used for the evaluation. Baseline parallel implementations for all of them were in Pthreads (bzip2 is the pbzip2 implementation in [13]). We studied and characterized the prototype on three stock multicore machines. Table 2 lists their configurations. The first uses a single-socket, 4 core, 2-way hyperthreaded, Intel iCore-965 (Nehalem) processor. The other two use 4- and 8-socket, 4 core/socket AMD Opteron 8350/8356 (Barcelona) processors. The sequential versions of the benchmarks were first ported to C++ and then modified using the programming interface for execution with the model. It is noteworthy that our system required no modification to the software tool chain. We compiled the code using `gcc-4.3.2 -O3 -march=amdfam10` for the Opteron machines, and with `gcc-4.3.2 -O3 -march=core2` for iCore7.

Ease of Programming

Codes developed for the benchmarks using our model closely resembled the sequential versions intended to run on a uniprocessor. No threads were created and no synchronization primitives were used to facilitate concurrent execution. No explicit work distribution or recreation of the original execution sequence was required. Further, no pattern-specific algorithm structures [19] were needed to exploit parallelism. We elaborate below using bzip2 as an example.

The main loop of bzip2 in our model is shown in Figure 7a. Note the lack of synchronization constructs, and its similarity with the sequential version in Figure 7b, but for the creation of write sets (lines 2 and 6 in Figure 7a) for the two functions, *compress* and *wr_file*, invoked through *df_execute*.

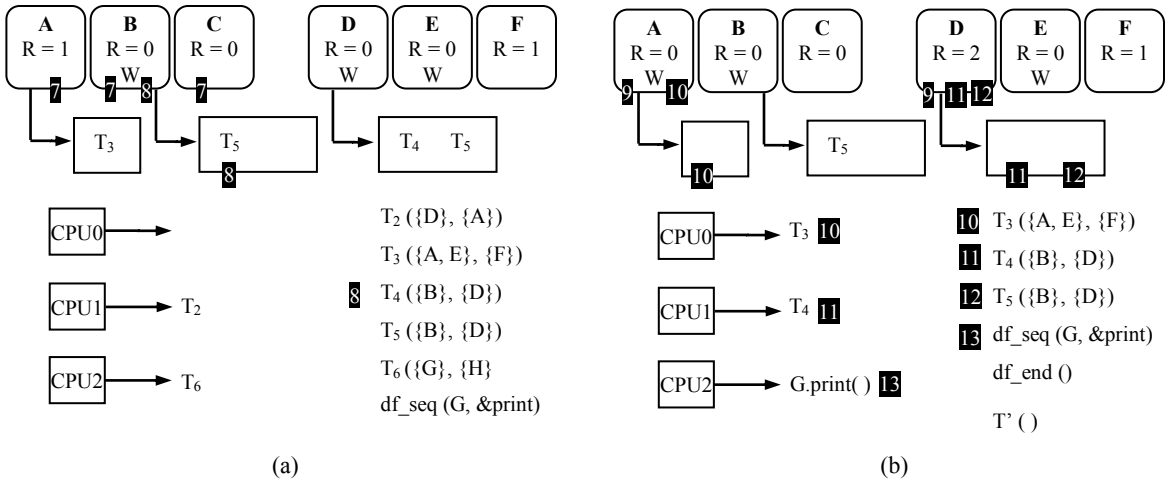


Figure 6. Example execution. (a) State of the system after invocation T1 completes execution. (b) State of the system after invocations T2 and T6 complete execution.

Table 1. Benchmark applications used for evaluation.

Benchmark	Inputs (Small/Medium/Large)
barneshut	(1,000, 25)/(10,000, 50)/(100,000, 75) (bodies, steps)
blackscholes	16, 384/ 65,536/ 10,000,000 options
bzip2	31MB/ 185MB/ 673MB file size
dedup	31MB/ 185MB/ 673MB file size
histogram	100MB/ 400MB/ 1.4GB bitmap size
reverse_index	100MB/ 500MB/ 1GB directory size

Table 2. Machine configurations used for experiments. (Core i7 = Nehalem; Opteron = Barcelona)

Machine Specs	Intel Core i7 -965	AMD Opteron 8350	AMD Opteron 8356
Sockets	1	4	8
CPUs per socket	4	4	4
Threads per CPU	2	1	1
Total contexts	8	16	32
Clock	3.2 GHz	2.0 GHz	2.3 GHz
L1 I$, D$ per cpu	32 KB	64 KB	64 KB
L2 $ per CPU	256 KB	512 KB	512 KB
L3$ per socket	8 MB	2 MB	2 MB
Total Memory	12 GB	16 GB	80 GB
Linux Kernel	2.6.18	2.6.25	2.6.25

By contrast, the Pthreads bzip2 is complex, as is also noted in [12], and too large to show here. It uses task-specific threads, one each for file read and write, and the rest to compress data. A pipeline of computations, based on the master/worker mechanism, is created by passing data (and hence work) from the file-reading thread to the compression thread, which in turn forwards its results to the file-writing thread. Threads communicate through lock-protected producer-consumer queues. To utilize resources efficiently in such a model, the user has to ensure just enough work is created for worker threads and idle workers do not consume cycles that may otherwise be utilized to perform work. In Pthreads bzip2 idle threads can suspend themselves and are awakened when work arrives or needs to be created, using wait-signaling. In our model, dataflow processing achieves these objectives without user intervention. It serializes dependent computations and schedules their execution when they are ready, thus automatically creating a pipeline of computations, e.g., by serializing a dependent file write after the computation but overlapping it with an independent computation. Further, lazy task creation and work-stealing ensure work is sought, and created if need be, by a context only when it is idle, thus efficiently utilizing resources.

Often the fork-join model (e.g., in Pthreads blackscholes), is used to achieve ordered file writes, underutilizing resources. To improve the utilization Pthreads bzip2 again relies on the master/worker model. Compressed data is logged into a predetermined slot in a lock-protected queue that is drained by another thread. In our model ordered execution of dependent computations made handling file I/O particularly straightforward, as shown in Figure 7a: invocations of the file-write function (line 8) simply serialize in the program order since they all write to the same object.

```
1 ..
2 op_set->insert (hOpfile); // File wr set
3 while (blockBegin < fileSize – 1) {
4   blockBegin += updateBlock (blockBegin)
5   block = new block_t (hInfile, Length);
6   block_set->insert (block);
7   df_execute (block_set, &compress)
8   df_execute (op_set,block_set,&wr_file)
9 }
10 ..
```

(a)

```
1 ..
2
3 while (blockBegin < fileSize – 1) {
4   blockBegin += updateBlock (blockBegin)
5   block = new block_t (hInfile, Length);
6
7   compress (block)
8   wr_file (hOpfile, block)
9 }
10 ..
```

(b)

Figure 7. (a) bzip2 kernel implemented using the proposed model (b) sequential bzip2 kernel.

We saw similar benefits for the other benchmarks. Finally, determinate execution resulted in repeatable programs and precluded the need to reason about the correctness of their parallel execution, considerably easing program debug and development.

Performance

Figure 8 shows the speedups achieved by our model and the Pthreads implementations available from the respective suites over the original sequential programs on the three machines. Speedups shown are for the large-sized inputs. They are based on the time taken to run the programs from launch to completion and include all runtime overheads and file I/O wherever performed. Ignoring the speedup bars marked '-LG' for the moment, compared to Pthreads, we achieve better speedups on dedup and reverse_index, while we do the same on histogram. We do worse on barneshut, blackscholes and bzip2. The harmonic mean (H_MEAN) shows we achieve 13%, 21.7% and 18.2% lower speedups than those achieved by Pthreads on iCore7, AMD 8350 and AMD 8356, respectively.

The Pthreads barneshut and blackscholes divide the data into equal-sized chunks and distribute them among threads to create coarse-grained tasks. However, for the above comparison, we used a grain size of only one object to retain the simple sequential programming style – the same as uniprocessor code, but at the cost of performance. In a second set of implementation we increased the granularity of the functions in barneshut and blackscholes to create 5 and 128 times as many chunks as the number of contexts in the system, respectively. We observed this yielded better performance than creating as many chunks as contexts since it permitted the runtime to better balance the load. For bzip2 we increased the file block size from 900K to 1.2MB. The bars marked '-LG' show improvements we achieved as a result. Barneshut being a data parallel application benefits the least from our model due to the absence of latent parallelism; our model incurs the overhead with limited benefit to the performance. Replacing the previous speedups for barneshut, blackscholes and bzip2 with the new figures, we now achieve a harmonic mean H_MEAN-LG (Figure 9) that is almost the same as that of Pthreads on the iCore7 machine, and better by 14.7% and 18.3% on AMD 8350 and AMD 8356, respectively.

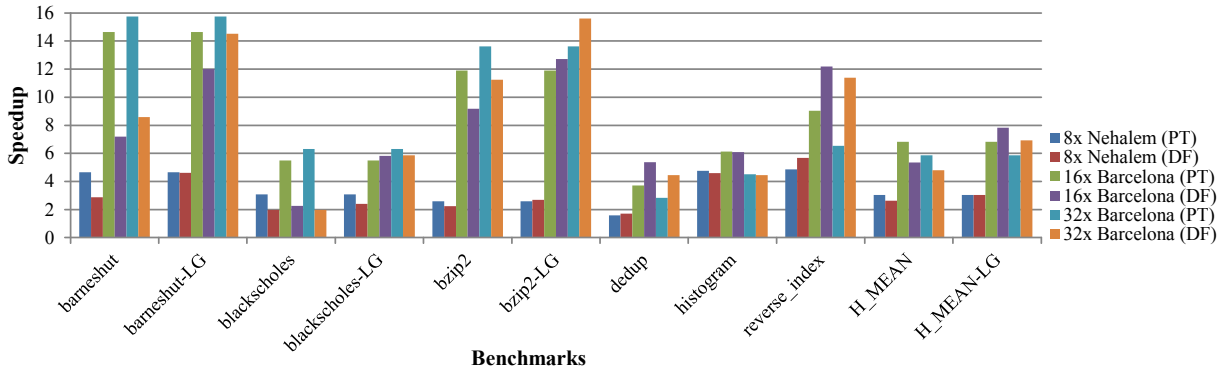


Figure 8. Speedups achieved by the proposed execution model (DF) and Pthreads (PT) on the three machines. LG = large grain; H_MEAN = harmonic mean.

The multi-socket Opteron machines have slower intra-socket buses than the single-socket iCore7 (1.8GHz vs. 3.2GHz), and even slower inter-socket buses (1GHz). They benefit more from chunking of data than the iCore7 due to the reduced communication-to-computation ratio and the reduced runtime communication overheads. Hence the relative performance gains of our model are higher on the Opterons than on the iCore7 for LG implementations.

Figure 9 shows the scalability of the execution model as input size changes, for the 16-core Opteron system. All applications but bzip2 and dedup scale with larger inputs. Bzip2 and dedup perform file compression and their performance is dependent on the input characteristic. In this case the medium sized inputs (same for both cases) achieved better compression ratio than the larger sized input, and hence the dip in scalability going from medium to larger size. All the scaling trends are similar to the Pthreads version (not shown).

Runtime Characterization

Being a software model, the prototype has multiple sources of storage and computational overheads as the runtime's interface and mechanics add indirect calls, manage tokens and wait lists, and shelve/schedule computations. We instrumented the runtime to measure the overheads and frequency of events in the prelude and postlude phases, and the execution time of functions, using microbenchmarks (described below) and the benchmarks in Table 1. Execution time for the microbenchmarks was measured in instructions while for the other benchmarks it was measured using the system-wide realtime clock via the Linux `clock_gettime()` function. For fine-precision analysis the overheads were measured in clock cycles. They were obtained by using the `rdtsc` instruction to read the x86 Time Stamp Counter. To ensure accuracy we first pinned the thread to the processor and flushed the pipeline before and after `rdtsc`.

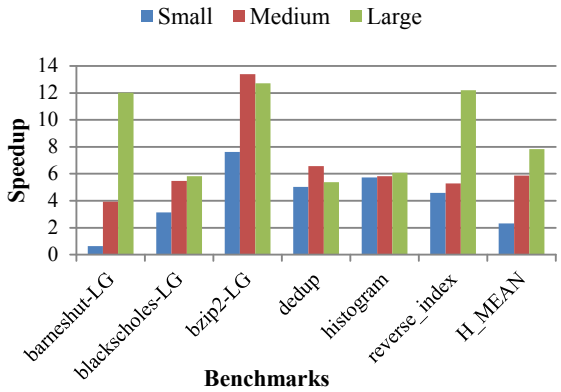


Figure 9. Performance scaling with respect to input size, for the 16-core Opteron

First, we used a microbenchmark to invoke dataflow functions with appropriate data sets and created various conditions leading to the different overheads. The overheads are machine dependent. Rounded off average measurements made on the three machines are listed in Table 3. We discuss the results below using the 16-core AMD 8350 data and refer to the rows in Table 3. Overheads are the same for both read and write tokens.

To study the bare bones token acquisition (Figure 4b, steps leading from 1 to 3) and release (Figure 4c: 1) overheads we created data sets such that the functions in the microbenchmark were independent. We varied the number of objects in the data set from 1 to 10 and measured the overheads for 10,000 such functions. On an average it took 225 cycles to acquire (Table 3,

Table 3. Results from microbenchmark evaluation: (i) Runtime overheads, in clock cycles per unit. (ii) Function granularity in instructions for profitable parallelization.

#	Overhead (cycles per unit)		Intel Core i7-965	AMD Opteron 8350	AMD Opteron 8356
1	Token Acquire (Figure 4b: 1 - 3)		225	225	200
2	Token Release (Figure 4c: 1)		200	225	200
3	Token Enqueue (Figure 4b: 1 - 4)		975	1100	2200
4	Token Pass (Figure 4c: 1, 2)	1 object	1000	1100	2000
5		>1 object	150	250	250
6	Prelude (Figure 3: 1)	Base	1100	1650	4000
7		Token Acquire	500	550	1000
8		Token Enq.	1150	1250	2950
9	Postlude (Figure 3: 3)	Base	500	500	750
10		Token Release	550	500	1000
11		Token pass(1)	1350	1400	2700
12		Token pass(>1)	150	250	250
13	Granularity (instr.)	Base	4000	12000	18000
14		Object	2000	2000	4000

row 1) or release (row 2) a token. To study the token enqueue overhead we created a pair of functions with the same write sets. Thus the second function was dependent on the first and every token request it made was enqueued in a wait list. We invoked 10,000 such pairs, varied the write set size from 1 to 10 and measured the overheads for the second function. The enqueue process attempts to acquire the token before logging the request in the wait list (Figure 4b, steps leading from 1 to 4). It allocates a data structure to hold the requester information and invokes race-free functions to log the request in the wait list, a concurrent queue [2]. Hence enqueue incurs a higher overhead of 1100 cycles (row 3).

To study the token passing overheads we created a set of functions that consumed (read) data produced (written) by a preceding function. Thus upon completion, the producing function passes tokens to the consumers. By varying the number of consuming functions we simulated token passing to up to 10 functions, and measured the overheads in the producing function for 10,000 such sets. Token passing first releases the token, traverses the wait list to pass it to waiting requesters, possibly enqueues ready requesters in the work deque (Figure 4c: 3), and deallocates data structures. Passing a token too involves operations on a concurrent queue and possibly the work deque. It incurs an overhead of 1100 cycles (row 4) for the first requester. In the current implementation, once the wait list traversal begins, concurrent queue processing simplifies, and hence only 250 cycles (row 5) are needed to pass tokens to every subsequent requester.

To characterize the total overheads of prelude (Figure 3: 1) and postlude (Figure 3: 3) phases, we used the same three set ups as before but measured the overall costs. The total overheads can be viewed as comprising of: (i) a base cost that is always incurred, and (ii) an additional cost proportional to the data set size. The prelude phase performs allocation and population of data structures, checks for duplicates in the data set and processes tokens. The total base overhead to delegate a function is 1650 cycles (row 6) and thereafter 550 cycles/object for an acquired token (row 7), or 1250 cycles/object for an enqueued request (row 8). Postlude releases tokens, deallocates structures and returns to the runtime scheduler. It requires 500 cycles in the least (row 9), and 500 cycles/object to return a token (row 10) thereafter. If a token is returned and passed, it consumes 1400 cycles (row 11) for the first requester, and 250 cycles for every subsequent requester (row 12).

Next we assessed the minimum task granularity needed to achieve speedups with the model. We invoked 10,000 functions from the microbenchmark and increased the total number of dynamic instructions in them until the parallel execution achieved lower execution time than the sequential version. We also varied the data set sizes from 0 to 10. With empty data sets, functions at least 12000 dynamic instructions long (measured counting actual executed instructions) are profitable to parallelize (row 13). Their size needs to grow by about 2000 instructions for every object added in the data set (row 14), to be profitable. The granularity results are similar to other task-based models [16].

Finally we analyzed the benchmark-specific characteristics of the runtime, using large-sized inputs. Resource management to limit program unfolding was not applied in these experiments. We present the data for the 16-core AMD 8350 system here in Table 4 (the other two machines showed similar trends). Column 1 shows the total number of functions delegated in each benchmark. Columns 2 and 3 give their average size in ms and clock cycles (Figure 3: 2), respectively. Of the total functions delegated, columns 4, 5 and 6 show the fraction that were shelved, the fraction that were delegated for execution while a function earlier in the program order was on a shelf (out-of-order), and the fraction that were delegated when no prior functions were on the shelf (in-order). Due to the very fine-grained tasks, blackscholes shelves relatively fewer functions, 5.86%, as compared to over 60%, mostly file writes, by blackscholes-LG, bzip2, bzip2-LG and dedup which use coarser-grain tasks, allowing following functions to proceed. Only one (the very first) function in blackscholes, blackscholes-LG, bzip2, bzip2-LG and dedup was executed in-order and remaining executed out-of-order. No functions needed to be shelved in the other applications, and hence all executed in-order. In case of reverse_index, file read overlaps with computations. A pipeline of block reads followed by computations is created. Each block is read in the main program context, the computation on it is delegated after the read completes, hence there are no shelved functions. Although large numbers of functions were shelved in the benchmarks, the maximum at any given time was only 80 (column 7), in bzip2. Column 8 gives the average number of functions that were on the shelf, when sampled at the end of every prelude phase (Figure 3: 1). The most were 26.9, also in the case of bzip2.

Column 9 shows the total number of tokens requested by each benchmark and column 10 gives the average size of write and read sets for each function in the benchmark. Columns 11 and 12 give the fraction of total tokens requested that were granted immediately upon request (TGOR) and passed to shelved functions in the waiting list (TPWL). Since only blackscholes, bzip and dedup shelve any methods, only they require any token passing, proportional to the number of shelved functions. Fine-grain blackscholes acquires over 90% of the tokens immediately, due to the small task sizes, while the coarser-grain blackscholes, bzip2 and dedup acquire only about 25% immediately. Functions in the other three benchmarks acquire tokens immediately.

Small data set sizes of the functions (column 10) resulted in quick token acquisition and release. Columns 13 and 14 show the average CPU cycles needed to acquire (Figure 3: 1.1) and release (Figure 3: 3.1) all tokens for a function. Compared to the average function size, 23us-307ms (column 2), the average token processing time is extremely small (125ns to 6.7us). Note that the token-release to token-acquire ratio is larger for benchmarks in which functions were shelved, since in these cases token return includes the overhead of passing the token, checking for and processing ready functions.

Tokens and wait lists can also impose storage burden on the system. Storage overhead of tokens is proportional to the number of shared objects. The maximum wait list size among all benchmarks was 80 (column 15), in case of bzip2 which delegated a total of 1566 functions. Note that although a total of 11.7M functions were shelved in blackscholes, only a maximum of 33 were shelved at any given instance. For barneshut, barneshut-LG, histogram and reverse_index the wait list sizes are zero since no methods are shelved. In all cases that shelve functions, except dedup, the maximum wait list size is the same as the maximum functions found on the shelf (column 7). In dedup, different functions shelve due to dependences on different objects, and hence no one list holds all shelved functions. In others, all shelved functions are dependent on at least one common object (the file pointer) causing its wait list to always hold all shelved functions

Table 4. Model characteristics on the 16-core Opteron: (1) Total functions delegated; (2, 3) Average function execution time in ms and clock cycles; (4-6) Fraction of functions shelved, executed out-of-order, and in-order; (7, 8) Maximum and average number of functions shelved at a time; (9) Total number of tokens requested; (10) Average write, read set sizes per function; (11-12) Number of tokens granted at the time of delegation (TGOR) and by passing to the wait list (TPWL); (13, 14) Cycles required to process token acquire and release; (15-16) Maximum and average occupancy of wait lists.

Benchmark	Function Processing								Token Processing							
	Total #	Avg Size (ms)	Avg Size (cycle)	On Shelf (%)	DF OO (%)	In Order (%)	Max on Shelf	Avg on Shelf	Total #	W,R per Fn	TGOR (%)	TPWL (%)	Req. (cycle)	Rel. (cycle)	Max WL Size	Avg WL Size
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
barneshut	7.5M	0.16	0.32M	0	0	100	0	0	7.5M	1, 0	100	0	1284	693	0	0
barneshut-LG	6000	119	238M	0	0	100	0	0	6000	1, 0	100	0	1430	2835	0	0
blackscholes	200M	0.02	0.05M	5.86	94.14	0	33	2	300M	1, 0.5	92.8	7.2	250	4696	33	0
blackscholes-LG	4096	231	462M	61.9	38.1	0	20	5.6	6144	1, 0.5	25.4	74.6	739	7835	20	0
bzip2	1566	251.3	503M	62.1	37.9	0	80	26.9	2349	1, 0.5	25.5	74.5	4099	13312	80	0.05
bzip2-LG	1174	307	2348M	62.4	37.6	0	62	24.6	1761	1, 0.5	25.3	74.7	4616	13194	62	0.07
dedup	1008	62	2016M	68.6	31.4	0	34	17.5	1344	1, 0.3	23.7	76.3	3558	14915	29	0.1
histogram	16	283	32M	0	0	100	0	0	16	1, 0	100	0	2255	4198	0	0
reverse_index	78356	0.53	1.06M	0	0	100	0	0	78356	1, 0	100	0	282	520	0	0

at any given time, and hence the parity in those cases. The average wait list size, sampled at the end of the prelude (Figure 3: 1) of every function, was less than 0.5 functions (column 16) for any application, indicating that wait lists are generally empty.

To track a shelved function the runtime maintains a data structure similar in size to an activation record of the function without the locals. With only a maximum of 80 functions shelved at any given time the storage overheads of wait lists and shelved functions are very reasonable.

When the access pattern of a group of objects is the same throughout the program, a single proxy object may be used to represent the group. Thus the runtime need manage only the proxy instead of each object in the group, thereby considerably reducing the total token and wait-list related overheads. For example, in blackscholes-LG and barneshut-LG data was chunked. We used a proxy object per chunk and identified it as the shared object.

5. RELATED WORK

The computing community has made much effort to extract function-level parallelism from programs. We divide the work into two broad categories based on the programming model employed, statically-parallel or statically-sequential.

A wide range of statically-parallel programming models have been proposed to exploit task/thread-level (TLP) parallelism. MPI, Pthreads, OpenMP, Cilk-5 [11], and TBB [16] are some of the more common interfaces. They use imperative languages to create parallel programs. The models can be deployed on a variety of CMPs or multithreaded processor platforms. Transactional Memory [15] has also been proposed to help ease programming by not requiring explicit synchronization. It ensures atomic execution of designated regions of code, which are executed speculatively. TM resorts to rollback if speculation fails. Unlike our model, in all of these solutions except TM, the onus is on the programmers or software tools to statically encode independence, synchronize accesses to shared data between parallel tasks, and reason about their correctness. If ordered execution is needed, such as for file I/O, user intervention becomes essential. Non-determinism is perhaps the biggest challenge in such models.

In another approach [7] a programming framework assisted by tools is proposed to extract parallelism from sequential code. However, it assumes a three-phase dependence pattern in loops and performs non-deterministic speculative execution of programs in contrast to our dataflow approach.

Proposals such as Multiscalar [23], Stanford Hydra CMP [14] and Program Demultiplexing [4] exploit speculative TLP from sequential programs on multicore architectures. These designs divide a program into tasks. Regardless of dependences, they speculatively execute tasks on different hardware contexts. Additional support is needed to track and recover from misspeculation. While our proposal also divides a sequential program into functions, it determines the data dependences between the functions and schedules them in a dataflow fashion, precluding the need for speculation and the concomitant support.

SMPSs [21] is a recent sequential program-based framework that achieves function-level dataflow execution. It requires the user to identify function parameters on which dependences may occur, using pragma directives. Although similar in philosophy, SMPSs builds a dynamic task-flow graph, during execution, based on memory locations accessed by them while we maintain an object-based dataflow graph. SMPSs renames data, potentially incurring high memory usage, whereas we employ the token protocol to handle WAW dependences. SMPSs uses a master thread to farm out work to other threads whereas we employ a decentralized scheduler. Since no characterization data is provided we are unable to compare performance artifacts. Task Superscalar [10], a more recent proposal is the hardware version of SMPSs. It emulates a superscalar processor by treating cores as execution units and adding additional eDRAM-based centralized structures analogous to register file, renaming table, reservation stations, queues, etc. We on the other hand map dataflow principles on existing multicores, and use distributed resources and mechanisms to account for the challenges posed by the scale. The almost direct correspondence to superscalar in Task Superscalar may be less apt

in multicores due to differences in granularities of computation and data sharing, need for data privatization, etc., necessitating a rethinking of the optimum hardware-software division. It is likely to be the subject of future research.

Deterministic Parallel Java (DPJ) [5], Jade [22] and Yada [12] provide language-specific extensions to help users write deterministic parallel programs by specifying access characteristics of shared data. DPJ performs compile-time type checks while Jade and Yada perform run time checks to ensure access types are not violated. We provide a runtime library for a standard language that ensures dataflow execution. Serialization Sets (SS) [1] is a sequential program-based, determinate model that dynamically maps dependent (independent) computations into a common (different) "serializer". Computations within a serializer are serialized while from different serializers are parallelized. Our model is similar to SS in philosophy and the programming interface. In fact, we build upon SS to achieve an even more dataflow-like execution and exploit higher degrees of concurrency.

6. CONCLUSION

In this work we have presented a novel execution model that achieves function-level parallel execution of statically-sequential imperative programs on multicore processors. Parallel tasks (program functions) are dynamically extracted from a sequential program and executed in a dataflow fashion on multiple processing cores using tokens associated with shared data objects, and employing a token protocol to manage the dependences between tasks. We thus combine the benefits of sequential programming and dataflow execution.

Rather than evaluate the model using simulation, we built a fully functional software prototype in the form of a runtime library. The paper described its architecture and implementation. We showed that benchmarks were easy to develop, and achieved performance similar to the traditional approach, despite the overheads of an all-software implementation.

Going forward, we expect the field to move away from traditional techniques, which require statically-parallel programs, towards techniques like the one proposed in this paper, to achieve the parallel execution of programs on multicore processors. We believe that this will have a significant impact on how multicore processors are architected and used in the future.

7. ACKNOWLEDGMENTS

This material is based upon work supported, in part, by the National Science Foundation under Grant CCF-0963737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Authors thank Matthew Allen and Srinath Sridharan for discussions related to this work.

8. REFERENCES

[1] Allen, M.D., Sridharan, S., and Sohi, G.S. Serialization sets: a dynamic dependence-based parallel execution model. *PPOPP*, (2009), 85–96.

[2] Allen, M.D. *Data-driven Decomposition of Sequential Programs for Determinate Parallel Execution*. Doctoral Thesis. Computer Sciences Department, University of Wisconsin-Madison, (2010).

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

[4] Balakrishnan, S. and Sohi, G.S. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. *ISCA*, (2006), 302-313.

[5] Bocchino, R.L., Adve, V.S., Dig, D., et al. A type and effect system for deterministic parallel Java. *OOPSLA '09*, (2009), 97-116.

[6] Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J., and Houston, K. *Object-oriented analysis and design with applications, third edition*. Addison-Wesley Profession, (2007).

[7] Bridges, M.J., Vachharajani, N., Zhang, Y., Jablin, T., and August, D. Revisiting the Sequential Programming Model for Multi-Core. *MICRO*, (2007), 69-84.

[8] Culler, D.E. and Arvind. Resource requirements of dataflow programs. *ISCA*, (1988), 141–150.

[9] Denning, P.J. and Dennis, J.B. The resurgence of parallelism. *Communications of the ACM 53*, 6 (2010), 30-32.

[10] Etsion, Y., Cabarcas, F., Rico, A., et al. Task Superscalar: An Out-of-Order Task Pipeline. *MICRO*, (2010), 89–100.

[11] Frigo, M., Leiserson, C.E., and Randall, K.H. The implementation of the Cilk-5 multithreaded language. *PLDI*, (1998), 212–223.

[12] Gay, D., Galenson, J., Nail, M., and Yelick, K. Yada: Straightforward parallel programming. *Parallel Computing 37*, 9 (2011), 499-652.

[13] Gilchrist, J. Parallel Data compression with Bzip2. http://compression.ca/pbzip2/.

[14] Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M., and Olukotun, K. The Stanford Hydra CMP. *MICRO*, (2000), 71–84.

[15] Harris, T., Larus, J., and Rajwar, R. *Transactional Memory, 2nd edition*. *Synthesis Lectures on Computer Architecture,* M. Hill, Ed., Morgan Claypool Publishers, (2010).

[16] Intel Thread Building Blocks: Reference Manual. Intel, (2011).

[17] Karp, R.M. and Miller, R.E. Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing. *SIAM Journal on Applied Mathematics 14*, 6 (1966), 1390-1411.

[18] Lee, E.A. The Problem with Threads. *Computer 39*, (2006), 33-42.

[19] Mattson, T.G., Sanders, B.G., and Massingill, B.L. *Patterns for Parallel Programming*, Addison-Wesley, (2005).

[20] Moshovos, A. and Sohi, G.S. Microarchitectural innovations: boosting microprocessor performance beyond semiconductor technology scaling. *Proceedings of the IEEE 89*, 11 (2001), 1560-1575.

[21] Perez, J.M., Badia, R.M., and Labarta, J. A dependency-aware task-based programming environment for multi-core architectures. *2008 IEEE Intl. Conf. on Cluster Computing*, (2008), 142-151.

[22] Rinard, M.C., Scales, D.J., and Lam, M.S. Jade: a high-level, machine-independent language for parallel programming. *Computer 26*, 6 (1993), 28-38.

[23] Sohi, G.S., Breach, S.E., and Vijaykumar, T.N. Multiscalar processors. *ISCA*, (1995), 414–425.

[24] Sutter, H. and Larus, J. Software and the Concurrency Revolution. *ACM Queue 3, 6*, (2005), 54–62.