Combining Compile and Run-time Dependency Resolution in Data-Driven Multithreading

Samer Arandi, George Michael, Costas Kyriacou, and Paraskevas Evripidou

1Department of Computer Science, University of Cyprus, Nicosia, Cyprus \{samer,xeirwn,skevos\}@cs.ucy.ac.cy
2Department of Computer Engineering, Frederick University, Nicosia, Cyprus eng.kc@frederick.ac.cy

ABSTRACT
Threaded Data-Flow systems schedule threads based on data-availability, i.e., a thread can be scheduled for executed when its data dependencies have been resolved. Two approaches are typically utilized for resolving the dependencies:

- A compile-time approach that is efficient but cannot handle programs with runtime-determined dependencies
- A runtime approach that can handle such programs but incurs runtime overheads even when part of the dependencies can be determined at compile-time

In this work, we propose a technique that combines the two approaches. The compiler attempts to resolve all the dependencies and encodes them into the Data-Flow dependency graph. For any unresolved dependency, it generates a helper thread that resolves the dependency at run-time and updates the graph accordingly with the help of an I-Structure. Thus, taking advantage of the strengths of both compile-time and run-time dependency resolution. The technique can be also utilized to improve the programmability and enhance the yield of compilation methods generating data-flow code.

In this paper we describe the proposed technique and present the implementation and evaluation details. The evaluation demonstrates that it is possible to obtain the benefits of the proposed technique without sacrificing the performance.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming; C.1.3 [Processor Architectures]: Other Architecture Styles—Data-flow architectures

General Terms
Design, Language, Performance

Keywords
Multi-core systems, data-driven multithreading, data-flow scheduling, I-Structures

1. INTRODUCTION
The shift to multi-core architectures had a fundamental effect on software, as it sparked the search for alternative programming models and tools that can exploit the large resources of multi-core chips efficiently. Part of the efforts in this direction incorporates data-flow [8, 5, 15] techniques (often at the level of scheduling) to take advantage of the distributed concurrency control mechanism of the data-flow model and its ability to tolerate latencies.

Data-flow based systems schedule tasks based on data-availability. A program is typically composed of a set of tasks or threads with consumer/producer dependency relationships. A task is deemed executable when all its inputs are ready, i.e., all its data dependencies have been resolved.

Two approaches are typically employed for resolving the dependencies: A compile-time approach and a runtime-time approach.

In the first approach, the dependencies are discovered at compile-time and statically encoded -typically as a graph. The graph is loaded -partially or fully- at runtime and used to schedule the tasks. This approach incurs the least overheads in terms of resolving the dependencies, as it only involves notifying -the already identified- consumers once a producer task finishes execution or once it produces a data value. However, some dependencies cannot be discovered at compile-time and thus the code has to be executed serially. Utilizing macro-based and directive-based methods can identify these dependencies but -in some cases- describing these dependencies is a hard task for the programmer, and -in others- not possible, if the dependencies can be only resolved at runtime. The Data-Driven Multithreading (DDM) [11, 13, 1, 2], which is the focus of this work, utilizes the compile-time approach.
The second approach, defers the discovering of dependencies to runtime. A task exposes its input and output data to a scheduler, which as the execution of tasks proceeds, examines the output data of a completed task and checks if it satisfies any of the pending dependencies. When all the dependencies of a task are satisfied the task is ready for execution. This approach can handle all programs but incurs extra overheads at runtime even when part of the dependencies can be determined statically at compile-time. Examples of system utilizing this approach include StarSSs [14, 12] and a number of the implementations of the Concurrent Collections (CnC) model [6, 7].

Data-Driven Multithreading (DDM) is a non-blocking multithreading model that combines the benefits of the dataflow model in exploiting concurrency with the highly efficient control-flow execution. The core of the DDM model is the Thread Scheduling Unit (TSU), which is responsible for the scheduling of threads at run-time based on data availability. DDM adopts a compile-time approach for discovering dependencies.

In this work we propose a technique that enables DDM to handle programs with runtime-time determined dependencies. We make use of I-Structures [4] for handling the synchronization between the producer and consumer threads in such programs in a split-phase manner, while allowing the compile-time dependencies to be utilized at the same time. The benefits of this technique extend to improving the programmability of the DDM-VM in the case of programs with complex dependencies, in addition to increasing the yield of compilation methods generating data-flow code when the dependencies cannot be uncovered at compile-time.

We implement the technique in the Data-Driven Multithreading Virtual Machine (DDM-VM) [1, 2], which supports DDM execution on multi-core architectures. We evaluate the proposed technique overhead using 3 benchmark applications. The evaluation indicates that given large enough threads granularities, it is possible to obtain the benefits of the technique while maintaining much of the performance.

2. BACKGROUND
2.1 Data-Driven Multithreading Virtual Machine

The Data-Driven Multithreading Virtual Machine (DDM-VM) is a parallel software platform that supports Data-Driven execution on conventional control-flow multi-core systems. The DDM-VM utilizes DDM scheduling for exploiting the resources of multicore architectures and tolerating synchronization and memory latencies. It employs data-flow concurrency for scheduling threads and efficient sequential execution of instructions within a thread. The scheduling of threads is orchestrated by the Thread Scheduling Unit (TSU), which is implemented as a software module running on one of the cores. The TSU is aided by the runtime that supports DDM execution on the rest of the cores. The work of the TSU is overlapped with the execution of the threads to tolerate the latencies and shorten the critical path of the application.

The DDM-VM has two implementations: The DDM-VM for the Cell [2] (DDM-VM\(_c\)), which targets heterogeneous multi-core architectures with software-managed memory hierarchies (the Cell processor [10] is the primary example of such architectures) and the DDM-VM for Symmetric Multicores (DDM-VM\(_s\)), which targets homogeneous multi-core architectures.

In the DDM-VM\(_c\) implementation, shown in Figure 1-a, the TSU runs on the general purpose Power Processor Element (PPE) core and the execution of the threads takes place on the Synergistic Processor Element (SPE) SIMD cores. A prefetching software cache (Software CacheFlow) is employed for managing the memory hierarchy of the Cell. In the DDM-VM\(_s\) implementation the TSU runs on one of the cores and thread execution takes place on the rest of the cores. This implementation is shown in Figure 1-b.

DDM-VM Programming. DDM-VM programs are partitioned into a number of DDM threads with each thread meta-data encoded as a synchronization template. Each thread is identified by the ThreadId and context. The context corresponds to the tag in dynamic data-flow and uniquely identifies dynamic invocations of each thread. The basic information specified in the synchronization template includes:

- The Instruction Frame Pointer (IPF): points to the address of the first instruction of the thread.
- The ReadyCount (RC): a value equal to the number of producers for this thread.
- The Data Frame Pointer List (DFPL): a list of pointers to the thread inputs/outputs data.
- The Consumer List (CL): a list of the thread consumers that is used to determine which RC values to decrement once the thread finishes execution.

All the threads templates constitute the synchronization graph describing the consumer-producer dependencies amongst the threads. The graph is loaded into the TSU at program startup and used for scheduling the threads based on data-availability. When a thread finishes execution the TSU uses the graph to identify and decrement the RC of its consumers. When the RC of a specific consumer reaches zero, it means that all its required data is produced. If prefetching is supported, the thread data is prefetched from main memory to the cache of the processor where the consumer thread is scheduled to run and the thread can be started.

Programming Toolchain. The following two approaches are available for programming the DDM-VM:

1. Macro-based: this is the low-level interface for programming the virtual machine. It utilizes a set of C macros, which expand into calls to the DDM-VM. The rest of the approaches target this interface.
2. T-Flux preprocessor: utilizes the TFlux directives and preprocessor tool originally developed in [13]. A subset
of the directives is extended to generate the DDM-VM macros.

In addition, two compiler projects are currently under development to further facilitate the programming of the DDM-VM. The first utilizes a GCC-based auto-parallelizing compiler that automatically generates code targeting the DDM-VM. The second is based on Concurrent Collections (CnC) [6, 7], which is a platform-independent, high-level parallel language. This project utilizes a compilation tool that parses CnC code to generate the DDM-VM macros.

For a detailed description of the programming methodology utilized with the DDM-VM please refer to [1].

2.2 I-Structures
An I-Structure [4] is a type of storage controller that obeys the single-assignment rule: Each element is written only once but can be read multiple times. If a read request arrives for a storage element that has not been written yet, the controller defers the read until a write arrives [3]. This property of I-Structures provides the synchronization needed for exploiting producer-consumer parallelism without the risk of read-write races [3]. Our technique utilizes the same property to discover the producer-consumer dependencies at runtime.

The basic idea in I-Structures is to add status bits to the storage cells in addition to a queue for holding deferred reads. The status of each element or storage cell of the I-structure can be:

- present: the cell data is valid and can be freely read but any attempt to write to it will be considered an error.
- absent: nothing has been written into the cell yet and no attempt has been made to read it. A write operation is allowed.
- waiting: nothing has been written into the cell yet, but at least one read request was attempted (deferred read). When this cell is written all the deferred reads must be satisfied.

A read operation on an I-Structure is commonly referred to as an I-Fetch and a write operation as an I-Store [4]. Figure 2 illustrates the state transitions of the I-Structure cells.

3. Run-Time Dependency Resolution with I-Structures
Assuming a program has been partitioned into a number of DDM threads. A problem arises when part or all of the threads perform read operation(s) on data items whose address is resolved at runtime. Although, other threads in the...
program potentially produce such data items, because the address of the consumed data is only determined at runtime, establishing the producer/consumer relationships amongst the threads is not possible at compile-time. To solve this problem we propose the following technique:

- For every thread \( t \) that performs at least one read operation on data items which address is resolved at run-time a proxy thread \( t' \) is introduced. Thread \( t' \) replaces \( t \) in all the Consumer Lists of its explicit producer threads, i.e., threads that are identified as its producers at compile-time.

- The RC of \( t' \) is set to the number of explicit producers of \( t \). The RC of \( t \) is set to the number of read operations performed on data items with run-time resolved addresses.

- For every such read a special \( I\text{-Fetch} \) request is issued by \( t' \). The first parameter of the \( I\text{-Fetch} \) is the address of the data to read, the second and third parameters are the thread identifier and context of \( t \). When the \( I\text{-Fetch} \) request is executed at runtime, it checks the I-Structure for the data address, if the address exists, i.e. the data has been produced, a request is sent to the TSU to decrement the RC of thread \( t \). If the data was not produced yet, the request is added into a pending list inside the I-Structure. Note that the \( I\text{-Fetch} \) in this manner is faithful to the non-blocking property of DDM, i.e. it doesn’t cause a wait by the issuing thread.

- For every thread producing data that is potentially read by \( t \), a special \( I\text{-Store} \) request is issued when the thread finishes execution. The \( I\text{-Store} \) registers the address of the produced data in the I-Structure, which results in sending all the existing pending requests on that address to the TSU.

It is important to note that in contrast with the traditional usage of I-Structures, we only keep the addresses of the data in the I-Structure storage cells and the data itself resides in the conventional memory.

We explain this technique further with the help of an example.

3.1 Example

Figure 3 illustrates a synthetic example of a simple DDM-VM program composed of five thread. In the upper part of the figure thread \( T_5 \) consumes data items \( M \) and \( N \) produced by threads \( T_1 \) and \( T_2 \), respectively. In addition, \( T_5 \) consumes an element of the array \( A \), however, the exact address of the element (its array index) is determined at runtime. Array \( A \) elements are produced by threads \( T_3 \) and \( T_4 \) and so it is only at runtime that the producer-consumer link between one of the two threads and \( T_5 \) is established. The problem is to ensure that \( T_5 \) executes only after the array element it requires has been produced by either \( T_3 \) or \( T_4 \). The “?" symbol in the figure is used to indicate a value that is determined at runtime.

The lower part of the figure shows how this problem is solved using the proposed technique. A new proxy thread \( T_5' \) is introduced and its RC is set to two. \( T_5' \) replaces \( T_5 \) in the Consumer Lists of \( T_1 \) and \( T_2 \). An \( I\text{-Store} \) operations is added at the end of \( T_3 \) and \( T_4 \) to register the address of the produced element of array \( A \). Moreover, an \( I\text{-Fetch} \) operation is added to \( T_5' \). When \( T_1 \) and \( T_2 \) finish execution, they notify the TSU, which decrements the RC of \( T_5' \) twice making it zero. Consequently, \( T_5' \) executes, it evaluates the address of the requested element in \( A \) and issues an \( I\text{-Fetch} \) on that address. If the address is found in the I-Structure (an I-Store with the address of that element was executed previously by \( T_3 \) or \( T_4 \)) a request to decrement the RC of \( T_5 \) is immediately sent to the TSU and once processed, \( T_5 \) RC becomes zero and it runs. If the address was not found, the request is enqueued in a pending list waiting to be sent to the TSU once the corresponding \( I\text{-Store} \) operation occurs. Table 1 lists the two DDM-VM macros implementing the \( I\text{-Fetch} \) and \( I\text{-Store} \) operations in addition to the macros needed for initializing and cleaning up the I-Structure.

3.2 The I-Structure Implementation

The efficiency of the I-Structure implementation is central to this technique. In particular, the operation of finding the entry corresponding to an arbitrary address, which incurs a non-trivial overhead. One solution is to implement this search operation as a hashmap search. A similar approach was used to implement the search in the I-Structure software cache described in [16]. This solution is a general one that can handle any type of accessed data (e.g. scalars, arrays, lists, etc.) including recursive and dynamically allocated data, as each I-Structure entry is associated with an arbitrary address. On the other hand, if the accessed data consists of arrays of a pre-determined size, it is possible to utilize a different I-Structure organization consisting of an array of preallocated entries, each corresponding to one array element. This removes the overhead of the associative search on the expense of extra memory storage. In this work we explore the first approach as it is more general.

The low-level design of the software I-Structure depends on the implementation of the DDM-VM. In the DDM-VM, due to the limited size of the Local Store (LS) memory on the SPE cores executing the threads and the sharing of the same I-Structure across all the SPEs, the I-Structure is allocated in main memory. Consequently the \( I\text{-Fetch} \) and \( I\text{-Store} \) operations are executed by the general purpose PPE core running the TSU and so there is no concurrent access to the I-Structure hashmap, which makes the design simpler. In the case of the DDM-VM, however, the \( I\text{-Fetch} \) and \( I\text{-Store} \) are performed by the runtime threads and so concurrent access to the hashmap occurs. Although this complicates the design, it also permits a distributed implementation of the search operation on the hashmap.

Hopscotch Hashing algorithm. To implement the I-Structure we used the Hopscotch Hashing algorithm [9] because it is highly-scalable, it outperforms existing hashing algorithms on both single-core and multi-core machines and most importantly it delivers good performance even when the hashmap is more than 90% full. This particular feature is very important in the case of I-Structures, as the addresses of produced
Figure 3: DDM-VM Program with Run-time Determined Dependencies

Table 1: DDM-VM I-Structure Macros

<table>
<thead>
<tr>
<th>DDM-VM I-Structure Macros</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVM_IFETCH(ADDR, THREAD_ID, CONTEXT)</td>
<td>Checks if ADDR exists in the I-Structure. If so a request to decrement the RC of the invocation with thread identifier THREAD_ID and context=CONTEXT is inserted in the AQ, otherwise the request is added to a pending list.</td>
</tr>
<tr>
<td>DVM_ISTORE(ADDR)</td>
<td>Registers ADDR in the I-Structure. Any pending requests on this address are served.</td>
</tr>
<tr>
<td>DVM_ISTRUT_INIT(SIZE, CORE_NUM)</td>
<td>Initializes the I-Structure and sets the initial size of the buckets and the number of cores the will access the structure.</td>
</tr>
<tr>
<td>DVM_ISTRUCT_SHUTDOWN()</td>
<td>Shuts down the structure and perform cleanup tasks</td>
</tr>
</tbody>
</table>
data are typically kept throughout the duration of the application execution, thus increasing the density of the hashmap. We have leveraged this hashmap in our I-Structure implementation and used the data address as the key to the stored hashmap entries. The entries have a presence field (performing a similar function to the presence bit of a traditional I-Structure) and a head pointer that holds a list of pending requests on the data address.

4. DISCUSSION
The proposed technique has two shortcomings: (i) the inevitable overheads of the I-Structure operations existing despite the optimized hashing algorithm (ii) and the fact that proxy threads must execute the part of the original thread code that evaluates the address of the accessed data, as this address is needed for the I-Fetch. Because this code is also executed by the original thread, an issue arises if the code is expensive to execute twice or altogether impossible (for example if it calls a random number generator).

To handle the first issue we employ an optimization that is applicable for the class of programs in which it is possible to know, by analyzing the program, how many times an I-Fetch will refer to a certain data address. For such cases we can assign a counter-value to the I-Structure entry associated with that address. The counter is decremented for every I-Fetch operation on that entry and once the counter reaches zero the entry is removed. This reduces the total number of hashmap entries, consequently improving the search performance and reducing the overheads. The counter value is assigned via the same I-Store registering the data address in the I-Structure.

To handle the second issue, we allow the code to be executed twice as long as it is possible and inexpensive. Otherwise, we execute it once at the proxy thread and channel any results required by the original thread as input data produced by the proxy thread.

The proposed technique has many advantages. Not only does this technique allow handling programs with run-time dependencies, but it does so while allowing compile-time dependencies to be utilized at the same time. Thus, obtaining the benefits of both approaches.

Further, the technique is very beneficial for compilation techniques that target the generation of data-flow code. Traditionally, when the compiler is unable to uncover the dependencies due to the existence of pointers, for example, it falls back to running the code sequentially. However, leveraging this technique, the compiler can fall back to generating parallel code and leave the discovering of dependencies to occur at runtime. Regardless of the involved overheads this alternative is expected to yield—in most cases—substantially better performance than running the code sequentially.

Finally, this technique improves the programmability of the DDM-VM. In the case of complex code with complicated dependencies, the programmer can utilize this technique to quickly develop and run the DDM-VM application without going through the most involving step, which is the dependency analysis. Dependency analysis can be incorporated later to encode the dependencies at compile-time for improving the performance. In fact, we envision an extension of this technique that records the discovered dependencies at runtime so as to provide feedback for guiding the uncovering of the dependencies and encoding them in the program.

5. EVALUATION
In this section we evaluate our technique using the DDM-VM s implementation of the DDM-VM.

5.1 Experimental Setup
The DDM-VM s runs on a 12 core machine composed of two Six-Core AMD Opteron Processors with 64KB L1 D-Cache, 64KB L1 I-Cache, 1MB unified L2 cache and 6MB unified L3 cache and a 32 GB of RAM. The cores run at 800 MHz and have the Ubuntu Linux 2.6.31 server edition as the O.S.

The benchmarks used in the evaluation feature 3 kernels widely used in scientific applications: The blocked matrix multiplication, the blocked Cholesky factorization and the blocked LU decomposition. The characteristics of the benchmarks are presented in Table 2. All the benchmarks work on dense single-precision floating-point matrices.

All of the benchmarks were coded in C using the macros approach and compiled using the GCC 4.4.3 compiler. All reported speedup results are relative to the execution time of the best corresponding (non-DDM) sequential code on one core.

5.2 Experimental Results
We study the effect of the overheads of the I-Structure operations on the performance by comparing 3 versions of the benchmarks.

- The first version utilizes the compile-time approach for resolving the dependencies in the program.
- The second version combines both approaches; part of the dependencies are resolved at compile-time and the rest are resolved at runtime (using I-Fetch and I-Store operations).
- The third version utilizes the runtime approach for resolving the dependencies.

We compare the performance of the 3 versions for various thread granularities (16x16, 32x32 and 64x64). The results are depicted in Figure 4. We refer to the 3 versions in the figures and the subsequent text as C-D, CR-D and R-D, respectively. Note that for the MatMult benchmark, only the first and third versions are available, as the threads in this program have one data dependency.

The results demonstrate that, as expected, the best performance is delivered by the version utilizing the compile-time approach (C-D), followed by the version utilizing the combination of the compile-time and runtime approaches (RC-D).

The results show that the performance loss (relative to the compile-time version) is higher for lower granularities and decreases as we increase the granularity. For example, when using 10 cores in the Cholesky application, the performance
Figure 4: Speedup comparison: runtime-determined dependencies (R-D) v.s. runtime & compile-time determined dependencies (RC-D) v.s. compile-time determined dependencies (C-D) approaches.

Figure 5: Execution time comparison: execution time using the runtime-determined dependencies approach v.s. the runtime-determined & compile-time determined dependencies approach normalized to the execution time using the compile-time determined dependencies approach.
loss when utilizing the runtime approach for all the dependencies (R-D) is 43% for the smallest granularity (16x16) compared to 13.6% for the largest granularity (64x64). When utilizing a combination of the two approaches (CR-D) the loss is 14.8% for the smallest granularity compared to 2.2% for the largest granularity. The same observation applies to the two other benchmarks. This is clearly demonstrated in Figure 5, which depicts the execution time of the R-D and RC-D versions normalized to the execution time of the C-D version.

The reason is that as we increase the granularity of the threads by increasing the size of the blocks the threads operate on, the total number of blocks decreases and so does the total number of thread invocations. Consequently the number of I-Fetch and I-Store operations decreases, thus reducing the overheads. Moreover, increasing the granularity of the threads amortizes the I-Structure operations overheads.

Overall, we find the results very encouraging as utilizing the proposed technique (for part or all of the data dependencies in the evaluated programs) achieves acceptable performance compared to the compile-time approach, whilst utilizing thread granularities in the range we normally utilize in DDM-VM programs.

6. CONCLUSION
In this paper we have proposed a technique that enables the Data-Driven Multithreading (DDM) model to combine compile-time and runtime approaches for resolving dependencies thus obtaining the benefits of both approaches. The technique can be also utilized to improve the programmability and has the potential to enhance the compilation methods that target generating data-flow code. We implement and evaluate the technique in the Data-Driven Multithreading Virtual Machine (DDM-VM). The evaluation indicates that for programs with large enough thread granularities the benefits of the technique can be gained without sacrificing much of the performance.

7. REFERENCES


