TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems

Kyriakos Stavrou
tsik@cs.ucy.ac.cy

Marios Nikolaides
cs04mn1@cs.ucy.ac.cy

Demos Pavlou
cs04dp1@cs.ucy.ac.cy

Samer Arandi
csp6as3@cs.ucy.ac.cy

Paraskevas Evripidou *
skevos@cs.ucy.ac.cy

Pedro Trancoso *
pedro@cs.ucy.ac.cy

Department of Computer Science, University of Cyprus
75 Kallipoleos Ave., P.O. Box 20537, 1678 Nicosia, Cyprus
* Members of the HiPEAC Network of Excellence from the EU FP7 program

Abstract

In this paper we present the Thread Flux (TFlux), a complete system that supports the Data-Driven Multithreading (DDM) model of execution. TFlux virtualizes any details of the underlying system therefore offering the same programming model independently of the architecture. To achieve this goal, TFlux has a runtime support that is built on top of a commodity Operating System. Scheduling of Threads is performed by the Thread Synchronization Unit (TSU), which can be implemented either as a hardware or a software module. In addition, TFlux includes a preprocessor that, along with a set of simple pragma directives, allows the user to easily develop DDM programs. The preprocessor then automatically produces the TFlux code, which can be compiled using any commodity C compiler, therefore automatically producing code to any ISA.

TFlux has been validated on three platforms. A Simics-based multicore system with a TSU hardware module (TFluxHard), a commodity 8-core Intel Core2 QuadCore-based system with a software TSU module (TFluxSoft), and a Cell/BE system with a software TSU module (TFlux-Cell). The experimental results show that the performance achieved is close to linear speedup, on average 21x for the 27 nodes TFluxHard, and 4.4x on a 6 nodes TFluxSoft and TFluxCell. Most importantly, the observed speedup is stable across the different platforms thus allowing the benefits of DDM to be exploited on different commodity systems.

1 Introduction

Data-Driven Multithreading (DDM) [5] is a parallel model of execution that has its origin on the Dynamic Dataflow [1] model of computation. The adoption of Dynamic Dataflow techniques has an advantage as it exposes the maximum available parallelism. DDM follows the execution paradigm of Dataflow, i.e. instructions are scheduled for execution only when their input data have been produced. While applying dataflow scheduling at the instruction level is known to require a large amount of hardware resources [4], DDM overcomes this limitation by applying this policy at the level of sequences of instructions named Data-Driven Threads (DThreads). Within a DThread instructions are executed in a control-flow manner, exploiting any optimizations offered by the executing processor.

In addition to the potential performance benefit, the DDM model also offers a programmability advantage as for most programs it is relatively easy to identify the code of the DThreads, as well as the data dependencies among them. The DThread’s code and their dependencies are the only requirements for the DDM model, which is then able to handle the data transfers and synchronization implicitly.

In this work we propose TFlux, the Thread Flux System. TFlux most relevant contribution is the fact that it offers the DDM model to the user-level and at the same time provides a virtualization of the underlying system therefore, different hardware configurations can support the DDM model of execution transparently to the programmer. In addition, TFlux is a complete system implementation, from the hardware to the programming tools.

To achieve this goal, TFlux addresses virtualization at different levels. First, the TFlux DThread scheduling unit,
the TSU, is accessed through high-level calls to library routines. This indirectness allows for the implementation of this unit to be in hardware or in software. Second, the TFlux runtime system is implemented at the user-level, therefore allowing for the application to run on top of any commodity OS. This allows for the system to execute DDM and non-DDM applications simultaneously. Third, the identification of DThreads’ code and DThreads’ dependencies is done by augmenting the high-level code with pragma directives.

TFlux offers a preprocessor that allows applications to be easily ported to TFlux just by augmenting ordinary C code with proper directives. The code produced by the preprocessor includes all necessary calls to the runtime system. As these calls are done at high-level, commodity compilers are used to generate the code. As such, it is possible to produce binaries for any ISA.

In this paper we present a proof-of-concept for the TFlux platform by executing different applications on systems with hardware and software implementation of the TSU. The former experiments are performed on a Simics-based full-system simulator of a Sparc multicore system (TFlux-Hard), while the latter by native execution on a commodity x86 homogeneous multicore system (TFluxSoft) and a Cell/BE heterogeneous multicore system (TFluxCell). The experimental results show that TFluxHard with 27 nodes achieved an average speedup of 21x for the target applications. Native execution on both TFluxSoft and TFluxCell achieved an average speedup of 4.4x on a 6 nodes system. These results indicate that TFlux is a scalable platform and can offer the DDM benefits to a multitude of systems.

2 Data-Driven Multithreading

Data-Driven Multithreading is a model of execution that achieves efficient parallel execution through dataflow-like scheduling.

DDM programs are composed of non-overlapping sections of code called Data-Driven Threads (DThreads) which can be of any size. A producer/consumer relationship exists between DThreads. The dependencies among the DThreads in a DDM program are expressed by its Synchronization Graph, the nodes of which correspond to the program’s DThreads while its arcs to data dependencies between them.

Scheduling a DThread for execution is done dynamically at runtime in a data-driven manner. i.e. only when all its producers have completed their execution. The instructions within a DThread are executed by the CPU in a control flow order and any optimization, either by the CPU at runtime or statically by the compiler, are exploited.

Data-Driven DThread scheduling is achieved with the help of the Thread Synchronization Unit (TSU). Prior to the execution of any DThread, the Synchronization Graph of the program is loaded into the TSU. In particular, for each DThread, the TSU manages the list of its consumer DThreads and its Ready Count, a value that indicates the number of its producer DThreads. When a DThread completes its execution it notifies the TSU, which in turn decreases the Ready Count value of its consumers. When the Ready Count of a DThread reaches zero it will be deemed executable. When a CPU becomes available, the execution of this DThread will be initiated. For a CPU to find the next DThread to execute, it queries the TSU, which replies with the identifier of one of the ready DThreads. In case no ready DThread exists, the TSU will force the CPU to wait.

To allow programs with arbitrarily large synchronization graphs, without requiring equally large TSU, DDM programs can be split into DDM Blocks. Each DDM Block contains a subset of DThreads of the original program and its maximum size, regarding the number of DThreads it contains, is defined by the size of the TSU. In addition to the application’s DThreads, each DDM Block has two other DThreads, the Inlet and Outlet DThread. The purpose of the former is to load the TSU with all metadata of the DThreads belonging to that Block whereas the purpose of the latter is to clear the allocated resources. When all the DThreads of a DDM Block complete, the Outlet DThread is executed. Upon completion of a Block, the Inlet DThread of the next block is loaded into the TSU thus allowing the execution to progress to that Block.

3 TFlux Portable Platform

TFlux does not refer to a specific implementation on a certain machine but rather to a collection of abstract entities that allow execution under the DDM model on a variety of computer systems. As such, the main objective of TFlux is to serve as a virtualization platform for the execution of DDM programs.

Figure 1 depicts the layered design of the TFlux system. In particular, the top layer, which is the one programmers use to develop DDM applications, abstracts all details of the underlying machine. DDM applications are developed using ANSI-C together with DDM directives [18]. The DDM directives are used to express the DThreads’ code and the dependencies among them. The program then passes through the TFlux Compilation Tool-chain which creates the executable binary. The binary invokes the operations of the “TFlux Runtime Support” allowing execution under the DDM model. The Runtime Support runs on top of an unmodified Unix-based Operating System and hides all DDM-specific details such as the particular implementation of the TSU. One of the primary responsibilities of the TFlux Runtime Support is to dynamically load the DThreads onto the TSUs and invoke all TSU operations required for DDM execution.
3.1 Runtime Support

The virtualization TFlux provides is mainly due to its Runtime Support. The Runtime Support executes on top of an unmodified Operating System and allows for the DDM execution to be interleaved with the execution of non-DDM binaries by means of simple OS context switch operations.

For the use of a regular OS for DDM execution, the Runtime Support has to satisfy two important requirements. First, when an application is executed in parallel in either a shared-memory or a distributed memory multiprocessor, the runtime has to provide a way for the different DThreads of the DDM application to access the shared variables used in the producer-consumer relationships. Secondly, the runtime has to provide an efficient way for the communication between the application and the TSU.

To meet these requirements, we designed a simple user-level process, the Kernel, which is described in the next Section. Notice that the Runtime Support for the DDM applications is embedded into the code at compile time. Therefore, each DDM binary is self-contained, requiring no extra software.

3.2 Kernel

The runtime support starts its execution by launching \( n \) Kernels, where \( n \) is the maximum number of DThreads that can execute in parallel in the machine.

Figure 2 depicts the operation of the Kernel. Its first operation is to transfer the execution to the address of the first instruction of the Inlet DThread of the first Block. At the end of that Inlet DThread, as well as of any other DThread, the control jumps again to the Kernel code and more specifically to the FindReadyThread loop. At that point the Kernel requests from the TSU the next DThread for execution. When finally the call to the TSU returns a ready next DThread, the execution control jumps to the first address of that particular DThread. If more than one ready DThreads exist the TSU returns the one which, based on its internal policy, is most likely to maximize the spatial locality. When a DThread starts its execution, it is not interrupted by the kernel and when it completes, it notifies the TSU and transfers execution back to the Kernel. The TSU will then execute the Post-Processing Phase for the specific DThread during which the Ready Count values of its consumers are decreased. The next step for the Kernel is to transfer execution to the next ready DThread.

If the TSU returns the address of an Outlet DThread execution is slightly different. In particular, the Outlet DThread, after deallocating the TSU resources, loads onto the TSU the Inlet DThreads of the next block with Ready Count equal to zero. As such, the Inlet DThread is the DThread that will be returned to the CPU. The Inlet DThread will load onto the TSU the metadata of the subsequent block and therefore allow execution to proceed to its DThreads.

As for the Outlet DThread of the last block instead of performing the TSU Load operation, it forces its Kernel to exit. As such, a DDM program finishes its execution when all its kernels execute all DThreads assigned to them.

3.3 TFlux Thread Synchronization Unit

In the previous implementation of DDM, the D^2NOW [10], each processor needed to have its own private TSU due to the fact that its execution nodes were independent machines. In TFlux we decided to group the TSUs in a single unit named the TSU Group. The units of the TSU Group are split into two categories: those that serve the CPU that the TSU corresponds to and those that are common for all CPUs. The TSU Group comes with two major benefits compared to the solution where a distinct TSU exists for each CPU.

The first regards the TSU-to-TSU communication which is common during the execution of a DDM program. In the TSU Group this communication is handled internally.
without the intervention of any other unit.

The second benefit is that it is possible to provide the TSU functionality in software using only one execution entity that emulates the operations. As such, a multiprocessor can have the TSU functionality by devoting only one CPU.

3.4 Compilation Tool-Chain

The Data-Driven Multithreading C Preprocessor (DDM-CPP) [18] is a tool that takes as input a regular C code program along with DDM specific pragma directives and outputs a C program that includes all runtime support code and TFlux interface calls necessary for the program to execute on a TFlux architecture under the DDM model. This tool is logically divided into two modules, the front-end and the back-end.

The DDMCPP front-end is a parser tool which is independent of the TFlux implementation, i.e., its task is to parse the DDM directives and then pass the information to the back-end to produce the code corresponding to the target architecture. The back-end is built as the actions of a grammar for the DDM directives and is dependent on the target TFlux implementation. Its task is to generate the code required for the TFlux runtime support such as the Kernel code and the DThreads load operations to the TSU, among others.

4 TFlux Implementations

4.1 TFluxHard

TFluxHard is a shared memory Chip Multiprocessor where the TSU Group is implemented as a hardware unit. The TSU Group is attached to the system’s network as a memory-mapped device. A special unit, the Memory Mapped Interface (MMI), is responsible for snooping the network and transferring to the TSU all memory requests directed to it. When the TSU is to send a value to the CPU, the MMI signals the arbiter and when access is granted it writes the value onto the network. Figure 3 presents a TFluxHard chip configured with 4 cores.

![Figure 3. A TFluxHard chip with 4 cores.](image)

The CPU controls the TSU Group through specially encoded flags. At the TSU Group side these requests are decoded and trigger the appropriate TSU operation. The hardware requirement of the TSU is estimated to be approximately 430K transistors. This estimation was done using a methodology similar to the one presented in [16].

Grouping the TSUs in a single unit, as described in Section 3.3, is specially important for the TFluxHard implementation. In particular, as the TSU needs to be attached to the System Network in order to communicate with the CPU, having one TSU per processor would require a network able to support two connections per CPU. However, by grouping the TSUs in one unit, the TSU Group, only one additional connection to the System Network is required for the whole system, which significantly decreases the additional interconnection cost. For systems with very large number of CPUs it may be beneficial to have multiple TSU Groups. A version of the TSU Group supporting such functionality is currently under development.

According to our experimental results, the speed of the TSU Group, i.e., the time it needs to process the commands send to it by the CPU and perform the corresponding operations, has minimal impact on the execution time of DDM applications. More specifically, increasing this processing time from 1 to 128 CPU cycles, has less than 1% impact on the performance.

4.2 TFluxSoft

The TFluxSoft implementation is targeted to commodity multicore processors that support a single shared address space, and consequently a cache coherency protocol for the memory hierarchy included on the chip. An example of such processor is the recent Intel QuadCore [7]. Notice that while the implementation targets multicore systems, TFluxSoft may also be executed on a shared-memory multiprocessor.

In the case of TFluxSoft we implement the TSU as a software module that executes its code on one of the cores of the multicore processor. As this software module replicates the functionalities of what was implemented in hardware in both D²NOW and TFluxHard, it is named TSU Emulator.

![Figure 4. Example of the execution on TFluxSoft on a system with n CPUs. The last CPU is dedicated to the TSU Emulation process.](image)

For TFluxSoft, both the TSU Emulation and the execution of application’s DThreads use POSIX threads. The operations of the TSU Emulation are divided, some of them
are executed by the Kernels (Local TSU) while others are executed by the TSU Emulator (Figure 4). The code of the TSU Emulator is executed by an independent POSIX thread which runs on an available CPU.

The major issues for this implementation are related to the Post-Processing Phase (Section 3.2). When a DThread completes its execution, its kernel inserts the identifiers of its consumer DThreads in a shared unit named the Thread to Update Buffer (TUB). The TSU Emulator then reads the entries of the TUB and decreases the Ready Counts of the corresponding consumer DThreads. The Ready Count values are stored in a data structure named Synchronization Memory (SM). One such structure exists for each kernel.

To update the Ready Count value of one particular DThread, the TSU Emulator needs to locate the SM in which this DThread is stored in. As the number of nodes on the system increases, identifying the proper SM may be a costly operation as these units must be searched sequentially, one by one. Thread Indexing is a technique that allows the TSU Emulator to directly access the correct SM, consequently eliminating any unnecessary search operation. In particular, a special table which is automatically embedded into the application’s code by the DDM Preprocessor, the Thread to Kernel Table (TKT) associates each DThread with the SM containing its Ready Count value. As such, when the TSU Emulator is to update a DThread’s Ready Count, it can directly access the SM containing this DThread by consulting the TKT.

In order to insert the consumers of a completed DThread into the TSU a kernel needs to lock it for mutual access as, due to its shared nature, it is possible that another kernel is also writing into it at that particular time point. As each such access requires mutual exclusion, execution may stall often, and this idle time increases with the number of kernels. To avoid long idle periods the TUB is partitioned into segments. When a kernel writes into the TUB, it uses the first available segment using try.lock, a non-blocking technique which locks an entity only if it is available. Consequently, only one segment is locked by each kernel at any time point.

4.3 TFVuxCell

The Cell/BE [8] is a heterogeneous multi-core chip processor with nine cores, one general-purpose RISC processor called the PPE (Power Processor Element) and 8 fully-functional SIMD co-processors, each called the SPE (Synergistic Processor Element).

TFVuxCell runs the DThreads on the SPE cores, while the TSU is implemented as a software module, just as in TFVuxSoft, and executes on the PPE.

As in the TFVuxSoft implementations, each compute node has a corresponding TSU whereas the TSU internal structures are allocated in main memory. In the case of the TFVuxCell due to the particular characteristics of the Cell/BE processor, in addition to the TSU structures used in TFVuxSoft it is necessary to use another unit per TSU named the CommandBuffer which size is 128 Bytes. This unit, which is also allocated in main memory holds the commands sent by the kernels executing on the corresponding SPE. Also one shared buffer (SharedVariableBuffer) is used by all kernels for transferring the values of the shared variables between DThreads. Note that the addresses of these two buffers are the only information that a Kernel needs, in order to communicate with its TSU. The rest of the communication is done using the other fine-grain mechanisms provided by the Cell/BE, such as mailboxes and signals.

Whenever a DThread needs to notify its TSU of any event, it places a command into its corresponding CommandBuffer. Also, when a DThread completes its execution, the Kernel waits on a mailbox for the information about the next DThread to be executed, which is sent by the TSU Emulator running on the PPE.

The TSU Emulator, on the other hand, is in a loop checking the CommandBuffers of all Kernels and updates the internal status of each TSU based on these commands. Whenever a DThread becomes ready, the Emulator notifies the corresponding Kernel, i.e. by sending the identifier of this DThread through the SPE’s mailbox.

After a DThread completes its execution, the data produced is exported to the sharedVariableBuffer in the TSU Emulator address space in main memory. Later, and before a new DThread that consumes this data starts executing, this data is imported from the sharedVariableBuffer into the SPE Local Store (LS) memory space, where this new DThread will execute. This operation is performed using the DMA primitives.

5 Benchmark Suite and Experimental Methodology

For the performance evaluation of the different TFVux implementations, the experimental workload used consists of five different benchmarks. Two of them are custom designed kernels that regard common scientific operations [15], other two belong to the MiBench [6] suite, and one to the NAS [2] suite. We show their descriptions along with the different problem sizes tested in Table 1. The problem sizes are separated into three categories: Small, Medium, and Large. To avoid too short times for the native execution, for one of the benchmarks, MMULT, we needed to use larger problem sizes. We show this in the Figure by indicating the problem sizes for Simulated (S), Native (N), and Cell (C) systems.

The porting of the above benchmarks was performed using the DDM pragma directives [18]. The execution
Table 1. Experimental workload description and problem sizes.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Description</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Small</td>
</tr>
<tr>
<td>TRAPEZ</td>
<td>kernel</td>
<td>Trapezoidal rule for integration</td>
<td>S,N,C</td>
</tr>
<tr>
<td>MMULT</td>
<td>kernel</td>
<td>Matrix multiply</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N,C</td>
</tr>
<tr>
<td>QSORT</td>
<td>MiBench</td>
<td>Array sorting</td>
<td>S,N,C</td>
</tr>
<tr>
<td>SUSAN</td>
<td>MiBench</td>
<td>Image recognition / smoothing</td>
<td>S,N,C</td>
</tr>
<tr>
<td>FFT</td>
<td>NAS</td>
<td>FFT on a matrix of complex numbers</td>
<td>S</td>
</tr>
</tbody>
</table>

time measurements were collected using the Unix `time` shell command for the total execution of the application, and the `gettimeofday` system call to measure the execution time of the section of code that has been parallelized. The native execution time for the experiments depends on the input data set size and it ranges from 1 second up to 3 minutes approximately. While for the simulated architecture the results were collected with a single run, for the native execution, multiple runs were performed in order for the results to be statistically significant.

All the experimental results are reported as Speedup, which represents how many times a certain parallel execution is faster than the corresponding sequential execution. Notice that the baseline, i.e., the system used for the sequential execution, is a system with the same processor as the one used for the parallel execution. Also notice that the baseline program is the original sequential one, i.e., without any TFlux overheads.

For both the sequential and the parallelized versions of the benchmarks we evaluated variations with the basic loops being unrolled from 1 to 64 times. To calculate the speedup we used the variation that gave the minimum execution time for each of the parallelized and the sequential version of the program.

Finally, for the various multicore implementations, in order to avoid instabilities, we reserve a core for the execution of the Operating System.

6 Experimental Results

6.1 TFluxHard

6.1.1 Experimental Setup

To evaluate the hardware-based TSU implementation, we used a simulated machine which was built using the Simics [12] full-system simulator V3.0.12. The machine is based on a 28-core Sparc system named Bagle, executing Suse 7.3 Linux, kernel 2.4.14 SMP.

The memory hierarchy was modeled using Simics’ gcache modules which allow Simics to simulate and take into account the overhead of the MESI protocol. Each processor is configured with a 32KB primary data cache (L1 D$), 64B line size and 4-way set associative, and a 32KB primary instruction cache (L1 I$), 64B line size and 2-way set associative. The read latency for both primary caches is 20 cycles.

In addition to the base machine, the TSU Group, which is a hardware module for this implementation, was developed as a Simics module using the DML language offered by Virtutech for this purpose. Each access to the TSU is penalized with 4 additional cycles compared to a normal L1 cache access. However, as explained in Section 3.3, increasing this value to 128 cycles results in performance degradation of less than 1%.

6.1.2 Experimental Results

The experimental results that indicate the speedup for the different applications, number of Kernel configurations, and problem sizes are depicted in Figure 5. From these results it is easy to observe that for all cases the speedup increases for larger problem sizes. This is justified by the fact that as the benchmark’s execution time increases the parallelization overhead is amortized.

As for the per-benchmark results, these are justified as follows. TRAPEZ can be efficiently parallelized resulting in no DThread dependencies other than a reduction operation that is required at the end. In addition, TRAPEZ has very few data transfers between DThreads which allows it to achieve near optimal speedup. MMULT also achieves very good speedup. MMULT is an embarrassingly parallel application but suffers from a large number of coherency misses, limiting it from achieving the idealized speedup. In QSORT each DThread sorts one part of the array. At the end, these sorted sub-arrays are merged to produce the final one. This last phase is the bottleneck for this application as its execution time is comparable to that of the sorting operation. The current application is written with a two-level tree to do the merging. Trees of bigger depth would result in
higher parallelism but may not be always beneficial as the
number of steps would increase as well. SUSAN has three
distinct phases which have been parallelized independently,
the initialization phase, the processing phase and the one
during which the results are written to a large output array.
As in each phase the parallelism was efficiently exploited,
we managed to achieve very good speedup (24.8 on a 27
nodes system). As for FFT, this benchmark operates on
the data in phases, which can only be parallelized indepen-
dently. The limitation in the speedup comes from the fact
that there is an implicit synchronization overhead between
the phases.

The same benchmarks have been executed on a simu-
lated 9 cores X86 system similar to Bagle. The speedup
values observed and conclusions drawn are similar to those
reported in this Section. Unfortunately we are not able to
present the details due to lack of space.

### 6.2 TFluxSoft

#### 6.2.1 Experimental Setup

To evaluate the TFluxSoft architecture we used an IBM
x3650 server configured with 2 Xeon E5320 Quad-
Core processors. Each core has a 32KB, 64B line size, 8-
way set associative L1 data and instruction cache, which
have a 3 cycle latency. Each QuadCore has a 4MB, 64B
line size, 16-way set associative unified L2 cache, which
has a 14 cycle latency. The machine is equipped with 18GB
of DDR2 RAM clocked at 333MHz.

#### 6.2.2 Experimental Results

The performance results of the TFluxSoft native implemen-
tation are presented in Figure 6. Direct results comparison
between the two implementations, TFluxHard on Bagle and
native TFluxSoft, is not possible as the two systems have
different CPUs, different OS and different compilers. It is
easy to observe however, that the trends are the same.

The only benchmark that deserves further explanation is
QSORT. In particular, for the 2 and 4 CPUs case, increasing
the problem size does not always lead to better speedup.
This behavior is justified by the nature of the benchmark.
During the initialization phase of QSORT one CPU initial-
izes the array to be sorted. Parts of this array then need to
be transferred to the caches of the CPUs that execute the sort
operation. This leads to a trade-off. More data means larger
communication delay but at the same time, decrease of the
relative effect of the parallelization overheads. This behav-
or appears in the native TFluxSoft results only due to the
configuration of the execution machine. In particular, each
QuadCore CPUs organizes its four processors in two pairs
and our system has two such chips.

In TFluxSoft systems, executing a DThread comes with
a larger overhead compared to TFluxHard systems. This
is due to the need to invoke a number of TSU Emulation
functions when a DThread completes its execution. In con-
trast, for the TFluxHard systems invoking the necessary op-
erations is done by sending to the TSU Group appropri-
ate flags (Section 4.1) through simple memory store com-
mands. As such, to amortize the parallelization overheads,
the DThreads for TFluxSoft need to be of coarser-grain.
The experimental results show that for the TFluxHard the
best speedup can be reached even with small unroll factors
(2 or 4) whereas for TFluxSoft the loops needed to be un-
rolled more than 16 times.

### 6.3 TFluxCell

For the TFluxCell implementation, a Sony Playstation 3
(PS3) with Linux 2.6.23-rc1 SMP ppc64 OS and the IBM
Cell SDK V2.1 is used. The specifications for the Cell/BE processor of the PS3 machine are the same as the original Cell/BE processor with the difference that it has available to the programmer only 6 of the total 8 SPEs. One of the SPEs is turned off by the manufacturer in order to improve the production yield and another is reserved for OS security. This Cell/BE processor executes at 3.2GHz and the system is equipped with 256MByte XDR RAM for its main-memory.

Similarly to what was performed for TFluxSoft for the Cell/BE-based system we also tested different unrolling factors to check what the Thread size should be in order to overcome the overheads of the implementation. For this particular implementation, the unrolling factor required was shown to be larger than what was determined for the TFluxSoft implementation. For example for MMULT high speedup is only achieved with an unrolling factor of 64.

Regarding the speedup values, the results are presented in Figure 7. As in the previous Section, for each benchmark we show the results for Kernel configurations ranging from 2 to 6 and for each Kernel configuration we show results for the Small, Medium, and Large problem sizes.

From the results it is possible to observe that for TRAPEZ, MMULT and SUSAN the implementation achieves the high speedup values as expected. For QSORT the speedup for the Cell/BE implementation is lower than what was expected. This is due to the fact that the problem sizes tested were relatively small and therefore the overheads are not amortized. The reason for not using larger problem sizes is that they would not fit in each SPE Local Store. To overcome this limitation we would have to change the algorithm in order to perform the execution in stages, on different data each time.

7 Related Work

A number of projects are focused on exploiting the parallelism offered by the CMP architecture. In this section we present a representative subset of these projects.

In the Carbon [9] project, the authors propose augmenting the CMP with additional hardware queues to exploit data and loop level parallelism. The results are promising, as in the TFlux system, but the solution is not applicable to existing multiprocessors. In contrast, TFlux can achieve high performance without any modifications to the hardware or to the OS. In addition, Carbon requires extensions to the ISA and as such, modifications to the CPU cores and the compilation tool chain. TFlux however, requires neither of these.

Thread Level Speculation [14], implemented by the Stanford’s Hydra CMP, allows the programmer to break a sequential program into non-overlapping Threads without the need to statically identify data-dependencies among them. The hardware attempts to execute the Threads in parallel while tracking all memory accesses to detect dependencies. As such, parallelization of difficult programs is made easier. In case of miss-speculation Thread rollback/restart functions are activated. The Hydra CMP requires extra hardware which is reported to consume the real estate of a pair of L1 caches. TFluxSoft achieves similar speedup without the need for extra hardware; and as for the TFluxHard solution, this hardware is smaller and simpler. In addition, in the Hydra solution TLS targets only loop bodies and Subroutine calls whereas in TFlux parallelism can be of much finer grain. Currently in TFlux dependencies are statically defined but this issue is expected to be covered in future versions of our compilation tool chain.

The TRIPS [3] architecture is the first implementation of the EDGE instruction set which basic idea is to give the dataflow graph of the program ready to the hardware. The program is split into hyper-blocks, which are executed in a dataflow manner and execution applies sequential semantics across the hyper-blocks. TFlux follows the opposite direction, it keeps the sequential semantics within the DThreads, which resample the hyper-blocks and schedules the DThreads in a dataflow manner. Most importantly, TFlux manages its benefits using only fully commodity components whereas TRIPS requires significant redesign of the CPU cores.

Tiled architectures, such as RAW [11], SmartMemories [13], TRIPS [3] and WaveScalar [17] address several limitations of current processors like complexity, wire-delay and performance by replicating the basic, simple, tile across the chip. While this technique is a different way to exploit parallelism it requires complete redesign of the execution units and is not applicable to current systems. The target of TFlux however is different, to achieve high perfor-
mance using existing commodity components.

The previous implementation of the DDM model, D²NOW [10], was used to prove the ability of the model to deliver high performance and scaling without requiring any modifications to the CPU core. D²NOW used dedicated machines connected as a Network of Workstations. In order to support the DDM model, these machines were augmented with an additional hardware unit and executed the applications in dedicated mode without an Operating System. In addition, D²NOW programming was done at a low level as the code needed to be augmented with assembly instructions to perform the DDM scheduling.

8 Conclusions

In this paper we presented Thread Flux (TFlux), a complete system that supports DDM in a general way as it virtualizes any details of the underlying system. To achieve this goal TFlux has a runtime support that is built on top of a commodity Operating System, a Thread Synchronization Unit (TSU) which can be implemented either as a hardware or as a software module, and a preprocessor that, along with a set of simple DDM pragma directives, allows the user to develop DDM programs.

We have ported five benchmarks using the TFlux tools and tested the virtualization of TFlux by executing the same code on three different implementations. The first is a Simgcs based system with 27 CPUs where the TSU was implemented as a hardware module (TFluxHard). The second is a QuadCore-based system with 8 CPUs where the TSU was implemented as a software module (TFluxSoft). And the third is a Cell/BE-based system where the TSU was also implemented as a software module (TFluxCell).

The experimental results show that the performance achieved is close to linear speedup. In particular, the hardware based implementation achieves an average speedup of 21x on a system equipped with 27 nodes. As for the software based TFlux implementation (TFluxSoft and TFluxCell), an average speedup of 4.4x is achieved on a system with 6 node. In addition, and most important, from the results it is possible to observe that the results are stable across the different platforms thus allowing the benefits of DDM to be exploited efficiently on different commodity systems.

Acknowledgments

This work was supported by the Cyprus Research Promotion Foundation under Grand ΔΠ/0505/25/E.

References