Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 516240, 9 pages
doi:10.1155/2008/516240

Research Article

Stream Execution on Embedded Wide-Issue Clustered VLIW Architectures

Shan Yan and Bill Lin

Electrical and Computer Engineering Department, University of California, San Diego, La Jolla, CA 92093-0407, USA

Correspondence should be addressed to Shan Yan, shyan@ucsd.edu

Received 13 March 2008; Revised 11 July 2008; Accepted 9 November 2008

Recommended by Sri Parameswaran

Very long instruction word- (VLIW-) based processors have become widely adopted as a basic building block in modern System-on-Chip designs. Advances in clustered VLIW architectures have extended the scalability of the VLIW architecture paradigm to a large number of functional units and very-wide-issue widths. A central challenge with wide-issue clustered VLIW architecture is the availability of programming and automated compiler methods that can fully utilize the available computational resources. Existing compilation approaches for clustered-VLIW architectures are based on extensions of previously developed scheduling algorithms that primarily focus on the maximization of instruction-level parallelism (ILP). However, many applications do not have sufficient ILP to fully utilize a large number of functional units. On the other hand, many applications in digital communications and multimedia processing exhibit enormous amounts of data-level parallelism (DLP). For these applications, the streaming programming paradigm has been developed to explicitly expose coarse-grained data-level parallelism as well as the locality of communication between coarse-grained computation kernels. In this paper, we investigate the mapping of stream programs to wide-issue clustered VLIW processors. Our work enables designers to leverage their existing investments in VLIW-based architecture platforms to harness the advantages of the stream programming paradigm.

1. INTRODUCTION

Current and emerging applications in digital communications and multimedia processing, such as 3G cellular communications, wireless LANs, video streaming, and image analysis have extremely high processing demands. These applications employ algorithms that can easily demand tens of billions of operations per second. To achieve this rate of processing, processor architectures with a large number of functional units are required. Fortunately, the algorithms used in current and emerging applications typically exhibit an enormous amount of inherent parallelism, especially in the form of instruction-level parallelism (ILP) and data-level parallelism (DLP), that can be exploited to utilize a large number of functional units, as required to achieve the necessary processing rates.

In practice today in industry, the requirements of much higher design productivity have encouraged the use of embedded processors. In particular, embedded processors based on a VLIW processor architecture have become widely adopted as a basic building block in modern system-on-chip (SoC) designs. In addition to much higher design productivity and greater ease of design reuse as compared with custom-logic design flows, the programmability offered by programmable processors is often needed in many applications to facilitate different functions and to facilitate late design (or even postdesign) changes. Traditionally, conventional VLIW architectures have been based on a central register file organization. Sophisticated compilers that can effectively identify and exploit instruction-level parallelism are readily available for these conventional VLIW processors. However, it is well known that conventional VLIW architectures cannot scale to a large number of functional units. To facilitate concurrent read and write accesses to a central register file by all functional units simultaneously, every functional unit must inherently have its own dedicated memory ports to the central register file, which adds tremendous complexity to the address decoding and data multiplexing logic. Therefore, it is not practical to scale a conventional VLIW architecture beyond a handful of functional units (e.g., usually no more than 4 to 6 functional units).

To extend the scalability of VLIW-based architectures, considerable progress has been made over the years on *clustered* VLIW processor architectures [1–5]. Clustered VLIW architectures overcome the limitations of a central register file by dividing the functional units into *clusters*, each cluster with a small number of functional units, and providing each cluster with its own simple local register file. To access data that is located at another cluster's register file, the operands must be explicitly transferred by *copy* operations between cluster register files across a communication network. Depending on the number of clusters and the communication network architecture, the latency of the copy operation may be nonuniform to account for the separation distance of clusters. With the clustering of datapaths, wide-issue clustered VLIW architectures with upwards of 32 functional units [4, 5] are possible, making available an enormous amount of computational resources for arithmetic intensive applications.

The central challenge with clustered-VLIW architecture is the availability of programming and automated compiler methods that can fully utilize the available computational resources. Currently, most of the compiler research in the literature has mainly focussed on extending conventional scheduling methods to consider the impact of the spatial assignment of operations to clusters in order to maximize the available instruction-level parallelism [6–11]. Despite the progress in these space-time ILP-based scheduling techniques, it is well known that many applications do not have sufficient amounts of inherent instruction-level parallelism to effectively utilize a large number of functional units. Fortunately, in many digital communications and multimedia processing applications, the algorithms used often exhibit an enormous amount of inherent data-level parallelism that can be exploited to fully utilize a large number of functional units to achieve high processing rates. In particular, the available data-level parallelism is often in the form of *coarse-grain* computations that operate on *streams* of *independent* elements. However, conventional sequential programming languages like C have difficulties exposing this style of parallelism.

To exploit available DLP, vector-based SIMD extensions to VLIW processors have been developed (see [12] for citations), and autovectorization techniques have been developed to identify vectorizable code in traditional C programs (see [13] for citations). While effective, these SIMD extensions usually involve dedicated vector units that cannot be harnessed for nonvectorizable computations, and the autovectorization methods that target these SIMD extensions often require complex memory disambiguations and loop transformations if the source program is not written in a manner that directly exposes the available data parallelism.

To explicitly expose coarse-grained data parallelism, researchers have developed *stream programming models* that explicitly expose coarse-grain data-level parallelism as well as locality between coarse-grain computations [14–18]. A stream program consists of a collection of computation *kernels* that consume and produce elements from streams of data. The advantages of this decomposition are as follows: first, stream programs separate communication (the gathers and scatters of data to and from global memory) from the actual computation. Hence, bulk memory transfers can be scheduled ahead of the corresponding computation, thereby hiding the cost of the large memory latency that is becoming increasingly worse with the growing gap between DRAM and logic speeds. Second, stream programs expose producer-consumer locality by the intermediate streams that flow between computational kernels. By appropriate sequencing of kernels, the elements of an intermediate stream can be kept local in fast local memories rather than incurring the penalties associated with the writing and reading of the global memory. Finally, stream programs explicitly expose both data-level parallelism and instruction-level parallelism. Since each element of each input stream can be processed simultaneously and independently, multiple instantiations of the computation kernels can be invoked to process multiple elements simultaneously, thereby providing large amounts of coarse-grained data parallelism. Further, within each kernel, instruction-level parallelism is exposed since independent operations within a kernel can be executed in parallel. In contrast to vector programming styles where vector elements are usually limited to the basic types, stream elements can be arbitrarily complex user-defined record types with potentially many fields. In addition, whereas vector operations are usually limited to simple predefined operations-like vector addition and subtraction, stream programs can specify data parallel execution of arbitrarily complex user-defined kernels that can embody hundreds of operations.

To facilitate implementation of stream programs, researchers have also developed specialized stream processor architectures [19, 20] that explicitly provide hardware mechanisms for coarse-grained data-parallel execution of stream programs. However, unlike VLIW-based processors that have already been widely adopted in SoC designs, stream processor architectures as a general architecture platform are still in their infancy, and it is not yet clear if stream processor architectures will become widely adopted in embedded SoC applications. Moreover, the available computational resources in a stream processor architecture can only be effectively harnessed if the application is inherently coarse-grained data parallel through a hardwired SIMD-style instruction control mechanism. On the other hand, in a clustered VLIW architecture, all functional units are fully under the control of wide-instruction words. Hence, the parallel datapath clusters in a clustered VLIW architecture can perform data-parallel execution by executing the code that replicates the same operation over each datapath cluster via wide-issue instructions. On the other hand, for applications that have plenty of instruction-level parallelism, but without data parallelism, the computational resources can be harnessed through existing ILP-based programming and compiler methods. Furthermore, clustered VLIW architectures can be made *configurable* at design time to different issue widths, as appropriate for the processing demands of the target application, using the same ISA. Therefore, configurable clustered VLIW architectures can arguably support a much broader base of applications, leveraging existing and future investments in these architectures.

Although the suitability of VLIW-based architecture for data-parallel applications has long been recognized [21–23], automated compiler methods that target VLIW-based processors have been lacking for exploiting coarse-grained data parallelism effectively. This is in part because efficient coarse-grained data-parallel execution requires the careful orchestration of bulk memory transfers with large chunks of computations to hide or avoid the long latencies of global memory operations and to maintain a continuous feed of data to keep the parallel datapath clusters utilized. Unlike the stream programming paradigm where bulk memory transfers, producer-consumer locality, and computational kernels are exposed explicitly, extracting the same information from a conventional C program, if possible, would require complex program disambiguation. Other existing work relating data-parallel applications to VLIW-based architectures have focussed either on case studies [22, 23] or SIMD architecture extensions [24] that are in the direction of mechanisms found in stream processors.

In this paper, we attempt to enable the use of the stream programming paradigm to effectively capture and harness the available data parallelism found in many digital communications and multimedia applications, while taking advantage of the broad investments that have been made or are being made in VLIW-based processor platforms for embedded SoC designs. Our work attempts to address this issue by providing effective compiler methods for the efficient realization of stream programs on wide-issue clustered VLIW processors. Using our approach, developers can write applications with inherent DLP using a stream programming paradigm. For applications without inherent DLP, known space-time ILP-based schedulers can still be used with conventional C programs.

The remainder of the paper is organized as follows. Section 2 outlines our underlying architecture target. Section 3 presents the stream programming model. Section 4 describes our methodology for mapping stream programs to wide-issue clustered VLIW processors. Section 5 presents experimental results of the proposed methods. Section 6 summarizes related work. Finally, Section 7 presents concluding remarks.

2. UNDERLYING ARCHITECTURE

Our target architecture is similar to the Lx processor described in [4]. The Lx processor is a statically scheduled clustered VLIW architecture jointly designed by HP and STMicroelectronics. Each cluster in the Lx processor is a 4-issue VLIW core comprising four ALUs, two multipliers, one load-store unit, and one branch unit. Each cluster also includes 64 general registers. The first generation of the Lx architecture spans from one to four clusters (with corresponding 4-issue to 16-issue per instruction cycle). In our target architecture, we further expand the number of clusters to eight clusters by interconnecting the eight clusters over a two-tier communication network, as depicted in Figure 1. In particular, the eight clusters are organized into four quadrants, with two clusters per quadrant. Communication between the two clusters in the same quadrant incurs a communication delay of one cycle, whereas communication between clusters in different quadrants incurs a communication delay of two cycles. Moreover, we expanded the instruction word length to permit 7-issue per cluster, thus enabling all four ALUs, both multipliers, and the load/store unit to be used at the same time. Thus, our target architecture permits a maximum of 56 issues, 48 simultaneous arithmetic operations and 8 simultaneous load/store operations.

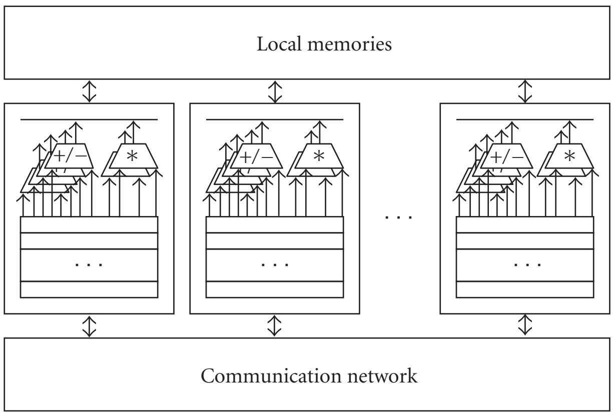

Figure 1: Target architecture.

Besides having a larger number of clusters than the first generation of the Lx processor, our target architecture uses *compiler-controlled* fast local memories rather than hardware-controlled data caches. Our target architecture uses a separate direct-memory-access (DMA) engine for transferring data between the global memory and the local memories. This DMA engine can perform bulk memory transfers *asynchronously*, under the control of our target processor.

3. STREAM PROGRAMMING

The stream programming model promotes a *gather-operate-scatter* style of programming for explicitly exposing coarse-grained data parallelism by incorporating the concepts of *streams*, *stream gather and scatter operations*, and *kernels*. A *stream* is a finite set of elements that can be independently operated upon in a data-parallel manner. *Stream gather* operations are used to gather data from different memory locations in the global memory into streams of independent elements. *Kernels* specify groups of operations, often several hundred operations in a kernel, that can operate in a data-parallel manner over the independent elements of streams. Streams can only be consumed and produced by kernels, and intermediate streams can be kept local in fast local memory. Coarse-grained data-parallel execution is achieved by executing multiple instances of a kernel in parallel by each instance operating on a separate element. Finally, output streams produced from a sequence of one or more kernels can be *scattered* back to different memory locations in the global memory via *stream scatter* operations. Stream gather and scatter

```
int x[300], y[300], z[100], t[100], f[400];
int i, m, p;

for (i = 0; i < 100; i++) {
    m = x[i + 200] * y[i + 50];
    t[i] = (m * m)/2;
}

for (i = 0; i < 100; i++) {
    p = t[i] + z[i];
    f[i + 300] = p;
}
```

Algorithm 1: Simple C code fragment.

```
(a) Kernel definitions

void kernel k1 (int sx⟨⟩, int sy⟨⟩, out int st⟨⟩)
{
  int m;
  m = sx * sy;
  st = (m * m)/2;
}

void kernel k2 (int st⟨⟩, int sz⟨⟩, out int sf⟨⟩)
{
  int p;
  p = st + sz;
  sf = p * 10;
}


(b) Stream operations and kernel calls

int x[300], y[300], z[100], t[100], f[400];
int sx⟨100⟩, sy⟨100⟩, sz⟨100⟩, st⟨100⟩, sf⟨100⟩;

streamRead (sx, x, 200, 100);
streamRead (sy, y, 50, 100);
streamRead (sz, z, 0, 100);

k1 (sx, sy, st);
k2 (st, sz, sf);

streamWrite (sf, f, 300, 100);
```

Algorithm 2: Stream program fragments.

operations essentially correspond to *bulk memory transfers* that can be scheduled ahead of or after the corresponding kernel computations, thereby hiding the long memory latencies. Moreover, dependencies in the form of streams flowing between kernels, or between stream gather-scatter operations and kernels, are explicitly exposed in a stream program. In essence, bulk memory transfers for streams and kernel computations over streams are decoupled in a stream program, thereby enabling their efficient orchestration. In particular, bulk memory transfers can overlap with kernel computations. For example, in the StreamC language [16], streams, gather-scatter operations, and kernels are modeled by means of a *signal flow graph*. In a stream programming language like Brook [14], streams, gather-scatter operations, and kernels are incorporated as language extensions to the conventional C language. *Streaming portions* of Brook can be easily extracted into signal flow graphs, for which stream program analysis techniques can be applied.

To illustrate the difference between the conventional programming paradigm and the stream programming paradigm, consider the simple C code fragment shown in Algorithm 1. There are two loops in this example. The first loop processes data from arrays x and y and writes the results into array t. The second loop processes data from arrays t and z and writes the results into array f. In the conventional programming model, the array reads and writes correspond to memory load and store operations. In general, array records need not be accessed in a particular order. In the example shown, the access patterns of the different arrays are different, in this case by different offsets. In general, array access patterns may correspond to irregularly accesses to the memory, which makes data-parallel execution more challenging as data must be loaded from potentially slow external memories and stored back to slow external memories. Although conventional processors provide data caches to mitigate the speed difference between external memory access speeds and internal processor computational speeds, caching often leads to unpredictable performance results. Without predictable data access speeds, it is very difficult to coordinate data-parallel execution of operations that must operate on the data.

In contrast to the conventional programming model, the stream programming model makes explicit the separation between memory operations and computational kernels. Using the Brook language [14], the stream program version of Algorithm 1 is shown in Algorithm 2. The corresponding extracted signal flow graph is shown in Figure 2. In Algorithm 2(b), the *streamRead* statements first copy data from arrays into *streams* from external memory so that the data-parallel records are readily available for the processor to perform data-parallel computations on them. Once the streams have been loaded, calls to *kernels* explicitly invoke data-parallel execution over the records of the streams. The intermediate streams produced, in this case *st* by the kernel $k1$, are consumed locally by other kernels, in this case by the kernel $k2$, without storing back to external memory. Final results are then written back to external memory in bulk via the *streamWrite* statement. As stated previously, stream gather and scatter operations can be scheduled ahead of or after the corresponding kernel computations, thereby hiding the long memory latencies, and bulk memory transfers can overlap with kernel computations.

4. WIDE-ISSUE CLUSTERED VLIW MAPPING

The focus of this work is on the mapping of a stream processor abstraction on to a wide-issue clustered VLIW architecture. In particular, Section 4.1 reviews a previously proposed stream processor abstraction called a stream virtual machine (SVM). The use of this virtual machine abstraction

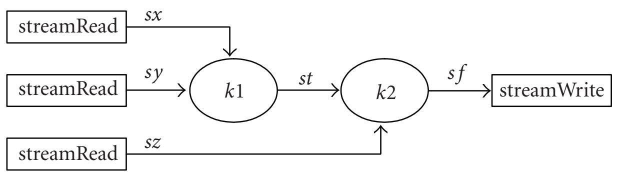

FIGURE 2: Stream program as a signal flow graph.

enables us to leverage previously developed stream compiler methods [16], which is described in Section 4.2. Section 4.3 then describes how the different components of the stream virtual machine are implemented on a wide-issue clustered VLIW architecture.

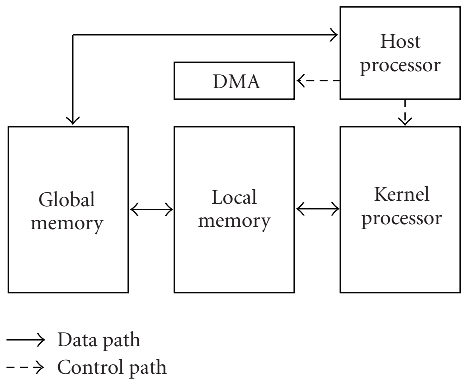

FIGURE 3: SVM architectural model.

4.1. Stream execution model

To improve the portability of stream programs on to different stream execution targets, a two-level compilation model was proposed in [25]. In this two-level compilation model, a stream program is first compiled to a Stream Virtual Machine (SVM), which is an abstract machine model that abstracts the detailed characteristics of the target processor. The SVM is then compiled, using target-specific compiler steps, to the target processor. The structure of the SVM model is depicted in Figure 3. It consists of three *virtual* execution engines and a two-level *virtual* hierarchy of memories. The three virtual execution engines are the *host processor*, the *DMA engine*, and the *kernel processor*, they work in the following manner.

(i) The host processor is responsible for scheduling bulk memory transfers on the DMA engine and scheduling kernel computations on the kernel processor. The DMA engine and the kernel processor are *slave* processors to the host processor.

(ii) The DMA engine is responsible for reading (gathering) stream data from global memory into local memory before kernel executions and writing (scattering) stream data from local memory to global memory after kernel executions.

(iii) The kernel processor is responsible for executing multiple instances of a kernel on independent stream elements in a coarse-grained data-parallel manner. The consumption and production of streams are via the local memory. The kernel processor does not access the global memory directly.

The two-level virtual hierarchy of memories provides a distinction between a slow (typically off-chip DRAM) global memory that can be randomly accessed and a fast (e.g., via on-chip SRAM) local memory that can only be accessed in a restricted manner. In particular, the local memory abstraction in the SVM is used to implement the *stream register file* (SRF), which is used to store input, intermediate, and output streams for kernel computations. There is a third level of memory hierarchy, not shown in the SVM diagram in Figure 3, that corresponds to the internal local registers inside the kernel processor. When executing a kernel in the kernel processor, intermediate values produced by operations in the kernel are stored temporarily in these local registers. Finally, the SVM is further characterized by the size of the memories and by the bandwidth and latency of the *network links* that interconnect the virtual execution engines and memories.

4.2. Compilation process

As proposed in [16, 25], the two-level compilation process consists of a high-level stream compiler step and a low-level detailed compiler step. The high-level stream compiler is responsible for extracting the signal flow graphs of streams, gather-scatter operations, and kernels. The extracted signal flow graphs directly expose data-parallel dependencies between gather-scatter operations and kernels. The high-level stream compiler is also responsible for the space allocation of the SRF and for the scheduling of the gather-scatter bulk memory transfers and kernel computations. In the SVM, the *size* of the local memory for implementing the SRF is specified. Typically, the size of the SRF is not large enough to store the input, intermediate, and output streams in their entireties. In this case, the high-level stream compiler is responsible for breaking down the streams into strips to ensure that each working set of strips can fit into the local memory and for scheduling the gather scattering of these strips and the kernel computations on them, a process referred to as strip mining.

In the second compilation phase, the low-level detailed compiler is responsible for the target-specific mapping of the three virtual execution engines and the two levels of memory hierarchy to the architecture components of the target architecture. In particular, the low-level detailed compiler is responsible for the actual instruction scheduling of the operations inside kernels and for the actual data-parallel execution of kernel instructions over the available computational resources. The low-level detailed compiler is also responsible for the detailed memory layout of the local memories for implementing the SRF, for the detailed scheduling of the actual bulk memory transfers between the global memory and the local memory, and for the actual detailed instructions for local memory access of stream elements for kernel computations. This low-level compilation phase is discussed next.

4.3. Target-specific mapping

Mapping the SVM to a wide-issue clustered VLIW processor involves mapping both the three virtual execution engines as well as the two-level virtual hierarchy of memories. Using our target architecture, some of the mappings are straightforward: the SVM global memory corresponds to an actual global memory, the SVM stream register file is mapped to the fast local memories, and the DMA engine is mapped to an actual DMA engine. The mapping of the host processor and the kernel processor is more challenging. In particular, the host processor and the kernel processor are combined into a *single thread of control*. This is possible because the kernel processor acts as a *slave* processor to the host processor. To combine the two virtual execution engines into one thread of control, we define two modes to represent the different statuses of the processor: *control mode* and *kernel mode*. In *control mode*, the processor executes the stream level control code. It initializes the DMA engine for memory access, responds to the interruption of the DMA engine when it finishes the data transfer, and executes all the other operations in the stream program except the kernel calls. All the clusters in the processor will work together as a normal wide-issue VLIW processor in this mode.

When the processor reaches a stream gather or scatter operation, it initializes the DMA engine by correctly setting the starting address and transfer count and starts the DMA engine. It also saves the state of the corresponding stream data. While the DMA engine takes care of the moving data between global memory and SRF, the processor can work on other tasks such as kernels whose input stream(s) are available in the SRF. After the completion of data transfer, DMA engine interrupts the processor, the processor will respond to the DMA after finishing the current cycle of execution. It then updates the state of the corresponding stream.

When the stream execution reaches a kernel call, it will do a context switch and change to the *kernel mode*. A specific portion of the local memory is reserved for saving the *control context*. During the context switch, the processor saves all the control registers as well as the current state information to the reserved region of the local memory. Then it loads the kernel code and switches to *kernel mode*.

In *kernel mode*, the processor works as the virtual kernel processor. As a wide-issue clustered VLIW processor, a large number of functional units (e.g., 48 functional units) are available. So our problem becomes how to efficiently utilize them to exploit the inherent parallelism inside the kernel. For the VLIW machine, all the instructions are decided at the compile time by the compiler. The clusters can be orchestrated and configured to work as many small VLIW machines to execute a separate instantiation of a kernel on each cluster. Data parallel execution is achieved by running k instantiations of a kernel on k separate stream elements simultaneously, in a load-balanced manner. For example, using a wide-issue VLIW processor with eight clusters, the clusters can work as follows: cluster 1 executes a kernel processing elements 1, 9, 17, and so forth, cluster 2 executes the same kernel processing elements 2, 10, 18, and so forth, cluster 3 executes processing elements 3, 11, 19, and so on. Since the execution is totally controlled by the very long instruction word, this way of data-parallel execution can be achieved.

To generate the detailed instructions for this data-parallel execution of the kernels, we first compile a kernel assuming one cluster. Then, for data-parallel execution of the kernel across all available clusters, the instructions generated for one cluster are replicated to all clusters to form the wide instruction words. The appropriate renaming is handled to form the wide instructions. In effect, each cluster is *looping* over every kth stream element, where k is the number of clusters. Because the stream elements can be independently processed, *software-pipelining* can be applied when compiling a kernel. For example, for cluster i, it will process elements i, $i + k$, $i + 2k$, and so forth. Software pipelining can be exploited by processing the $(i+k)$th element before the complete processing of the ith element, and so on. In particular, we have implemented the modulo scheduling technique in our kernel compiler [26, 27], extended with the spatial assignment heuristics proposed in [28].

After the completion of kernel execution in *kernel mode*, the processor returns back to the *control mode* by context switch. The control registers are restored and the state of the processor is updated.

5. EXPERIMENTAL RESULTS

We evaluate our mapping methodology on the target wide-issue clustered VLIW architecture outlined in Section 2, which contains 8 clusters, 6 FUs in each (4 ALUs, 2 multipliers), for a total of 48 FUs. For our evaluation, we used four popular applications in multimedia and communication, including two versions of the discrete cosine transform (DCT) algorithm, the AES encryption algorithm, and the AES decryption algorithm, which are applications that are inherently data parallel. The first version of DCT (pseudo-DCT) is a simplified pseudocode version of DCT used in [5]. It corresponds to a piece of DCT row. This version has limited ILP. The second version (optimized DCT) is from [29]. It uses an optimized 1D DCT as a kernel for the 2D DCT. This kernel executes eight 8-point 1D DCT on eight rows inputs and stores the outputs transposed. Each $8 \times 8$2D DCT is thus implemented by executing the kernel two times, one on row inputs, the other on transposed column inputs. This version has higher ILP. For the AES encryption and decryption algorithms, we used an RC6-32/20/32 block cipher [30], which encrypts and decrypts blocks of w = 32-bit words, using r = 20 rounds of iterations per block. To make them suitable for the stream programming paradigm and fair for comparison among different methods, we rewrote them by unrolling the $r = 20$ rounds into a single basic block. Then in the stream program, they are written as a single kernel. The encryption/decryption keys are first expanded in the stream program, and then they are passed to the kernels as constant parameters.

For comparison, we implemented 4 methods: the first two focussed on ILP exploitation; the last two based on the stream execution model for DLP exploitation.

(1) *List scheduling*. The focus of this method is on exploiting ILP. This method aims to find the maximum amount of ILP available within basic blocks to effectively utilize as many FUs as possible. In particular, we implemented a list scheduler that takes into consideration the spatial impact of cluster assignment. Our implementation is based on the space-time list scheduler method described in [6].

(2) *Modulo scheduling*. The focus of this method is also on exploiting ILP. This method aims to find the maximum amount of ILP available by pipelining across loop boundaries to effectively utilize as many FUs as possible. Our implementation is based on the space-time modulo scheduler method described in [26], extended with the spatial assignment heuristic described in [28].

(3) *Stream scheduling*. In this method, we used the stream programming paradigm and the mapping methodology described in Section 4. For compiling kernels, we limited the scheduling to use just list scheduling. DLP is achieved by executing multiple parallel kernel instances across available clusters.

(4) *Stream + modulo scheduling*. In this method, we again used the stream programming paradigm and the mapping methodology described in Section 4, but here we used modulo scheduling instead to identify more ILP within a kernel. DLP is again achieved by executing multiple parallel kernel instances across available clusters.

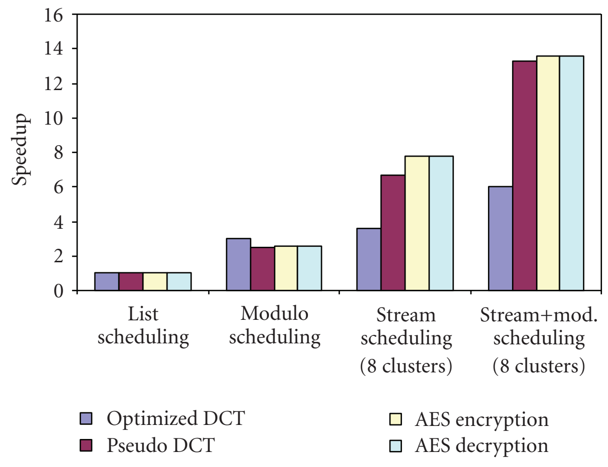

Figure 4: Execution time speedup results for the four benchmarks.

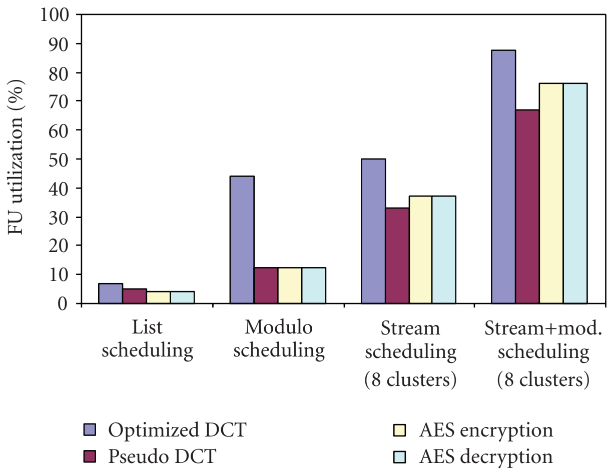

Figure 5: FU utilization results for the four benchmarks.

For the first two ILP-based methods, focused on ILP exploitation, the benchmarks were converted from a C program by hand to an internal control data flow graph (CDFG) representation. Then our implementations of list scheduling and modulo scheduling were applied. For the last two stream mapping methods, based on stream execution, the benchmarks were first written in Brook [14]. Then they were hand converted to an internal signal flow graph (SFG) representation. The kernels were further converted into an internal CDFG representation. Our implementations of stream scheduling of the SFG and the ILP scheduling of the kernels were then applied.

For comparisons, we use the list scheduling method as the baseline. Figure 4 compares the relative execution times of the four methods on the four benchmarks, and Figure 5 compares the FU utilization. For the applications that have little inherent ILP, such as pseudo-DCT benchmark and the AES encryption and decryption benchmarks, even with the use of modulo scheduling, most of the functional units cannot be efficiently utilized. The FU utilization is very low, only 5% for list scheduling and 12.5% for modulo scheduling for pseudo-DCT, and 4.34% for list scheduling and 12.36% for modulo scheduling for AES encryption and decryption. However, by exploiting DLP using the proposed mapping method, the degree of FU utilization goes up substantially, especially when combined with modulo scheduling for scheduling kernel operations. The FU utilization goes up to 67% for pseudo-DCT and 76% for the two AES benchmarks. Correspondingly, the execution times relative to list scheduling are substantially better: a 13.3X speedup for pseudo-DCT and a 13.6X speedup for AES benchmarks.

For the optimized DCT benchmark, there is more available ILP. With the available ILP in optimized DCT, our method of stream execution, combined with modulo scheduling for kernels, can achieve an FU utilization factor of 87.5%, versus under 10% for list scheduling alone. For this benchmark, modulo scheduling is able to find considerable ILP to effectively utilize the large number of FUs, nearly 45% FU utilization, but stream execution can still achieve a much higher rate of FU utilization. This is reflected in the relative execution times. Relative to basic list scheduling, modulo scheduling alone can achieve a 3X speedup. but stream execution with modulo scheduling can achieve a 6X speedup.

Note that we do not suggest by the above results that stream programs perform better than hand-tuned traditional programs. It is possible that existing autovectorization techniques [13] can be adapted to pure clustered VLIW processors without SIMD extensions so that a well-tuned traditional program can achieve similar results as our stream mapped programs. The above results mainly show that good data parallel execution performance can be readily achieved by mapping explicitly exposed data-parallel streamed programs on clustered VLIW platforms.

6. RELATED WORK

There has been substantial research in the literature previously on the extension of previously developed ILP-based scheduling algorithms to take into consideration the impact of spatial assignment of operations on execution times for clustered VLIW architectures [6–11]. In particular, in a clustered VLIW architecture, explicit copy operations are needed for moving values between clusters, the added nonzero latencies of these copy operations may adversely lengthen the critical execution path and degrade performance. Therefore, a poor spatial assignment of operations to clusters may result in a substantial degradation in processing rates. The consideration of spatial assignment of operations during scheduling has been referred to as the space-time scheduling problem. Most approaches investigated in the literature are based on extensions of commonly used scheduling algorithms, namely, list scheduling and modulo scheduling. However, these scheduling techniques are primarily targeted toward the exploitation and the maximization of the available instruction-level parallelism, which is often insufficient to fully utilize a large number of functional units. Our approach for the implementation of stream programs on clustered VLIW processors, which exploits data-level parallelism, provides a complementary solution to these ILP-based techniques. Our proposed solution is based on the two-level compilation and stream virtual machine model proposed in [25]. Our work provides the low-level mapping methodologies needed for efficient execution on a clustered VLIW processor.

Gummaraju and Rosenblum [31] recently proposed a low-level mapping solution for general-purpose CPUs like a Hyperthreaded Pentium 4. Though their problem bears similarity to our problem, their primary challenges are different. In particular, data caches in the Pentium 4 are not under compiler control. Therefore, a primary focus of their work is on the pinning of the cache to ensure that the stream register file will remain in the cache. In our problem, we provide mapping methodologies for using on-chip SRAMs that are embedded with the processor in an SoC design. Also, Pentium 4 is a superscalar processor with runtime dispatch of parallelizable instructions on a small number of functional units. In contrast, our problem is to efficiently exploit a large number of functional units (in the range of 32–48 functional units). To achieve high utilization, we must carefully at compile-time orchestrate the clusters to perform data parallel execution. In particular, we effectively execute a separate instantiation of a kernel on each cluster. Data parallel execution is achieved by running N instantiations of a kernel on N separate stream elements simultaneously, in a load-balanced manner. The mapping methodologies detailed in Section 3 provide the details for this data parallel mapping to parallel clusters. Finally, instead of using hyperthreading, our mapping methodologies are based on combining the control and kernel virtual processors into a single thread of execution on the clustered VLIW processor. If a separate DMA engine is not available, our mapping methodologies are based on combining the control, DMA, and kernel virtual processors into a single thread of execution.

Finally, although the suitability of VLIW-based architecture for data-parallel applications has long been recognized [21–23], automated compiler methods that target VLIW-based processors have been lacking for exploiting coarse-grained data parallelism effectively. This is in part because efficient coarse-grained data-parallel execution requires the careful orchestration of bulk memory transfers with large chunks of computations to hide or avoid the long latencies of global memory operations and to maintain a continuous feed of data to keep the parallel datapath clusters utilized. Unlike the stream programming paradigm where bulk memory transfers, producer-consumer locality, and computational kernels are exposed explicitly, extracting the same information from a conventional C program, if possible, would require complex program disambiguation. Other existing work relating data-parallel applications to VLIW-based architectures have focussed either on case studies [22, 23] or SIMD architecture extensions [24] that are in the direction of mechanisms found in stream processors.

7. CONCLUSION

In this paper, we explored the mapping of stream programs to wide-issue clustered VLIW processors. We showed how the explicit coarse-grained data parallelism presented in stream programs can be leveraged to harness the large number of functional units available in wide-issue processors. This way, designers can take advantage of their existing investments in VLIW-based architecture platforms to realize the benefits of the stream programming paradigm.

ACKNOWLEDGMENTS

We would like to acknowledge Alan Gatherer and David Hoyle from Texas Instruments, Inc. for a number of stimulating discussions on both application and compiler requirements for wide-issue clustered VLIW architectures. This work was sponsored in part by a grant under UC MICRO Program no. 2006-3139. Moreover, this work was presented in part as a poster at the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07), San Diego, Calif, June 13–15, 2007.

REFERENCES

[1] G. Slavenburg, S. Rathnam, and H. Dijksra, “The TriMedia TM1 PC1 VLIW media processor,” in *Proceedings of the 8th HotChips Conference*, pp. 171–178, Stanford, Calif, USA, August 1996.

[2] Texas Instruments Inc., "TMS320C6000: a high performance DSP platform," http://www.ti.com/.

[3] StarCore Alliance (Motorola Semiconductors and Lucent Technologies). Leadership in DSP technology for communications applications, http://www.starcore-dsp.com/.

[4] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: a technology platform for customizable VLIW embedded processing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 203–213, Vancouver, BC, Canada, June 2000.

[5] O. Colavin and D. Rizzo, "A scalable wide-issue clustered VLIW with a reconfigurable interconnect," in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 148–158, San Jose, Calif, USA, October-November 2003.

[6] V. S. Lapinskii, M. F. Jacome, and G. A. De Veciana, "Cluster assignment for high-performance embedded VLIW processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 430–454, 2002.

[7] C. Akturan and M. F. Jacome, "CALiBeR: a software pipelining algorithm for clustered embedded VLIW processors," in *Proceedings of International Conference on Computer-Aided Design (CAD '01)*, pp. 112–118, San Jose, Calif, USA, November 2001.

[8] J. Sánchez and A. González, "Instruction scheduling for clustered VLIW architectures," in *Proceedings of the 13th International Symposium on System Synthesis*, pp. 41–46, Madrid, Spain, September 2000.

[9] W. Lee, R. Barua, M. Frank, et al., "Spacetime scheduling of instruction-level parallelism on a raw machine," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, pp. 46–57, San Jose, Calif, USA, October 1998.

[10] G. Desoli, "Instruction assignment for clustered VLIW DSP compilers: a new approach," HP Technical Report HPL-98-13, Hewlett-Packard Laboratories, Palo Alto, Calif, USA, February 1998.

[11] R. Leupers, "Instruction scheduling for clustered VLIWDSPs," in *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, pp. 291–300, Philadelphia, Pa, USA, October 2000.

[12] E. Salami and M. Valero, "A vector-μ SIMD-VLIW architecture for multimedia applications," in *Proceedings of the IEEE International Conference on Parallel Processing (ICPP '05)*, pp. 69–77, Oslo, Norway, June 2005.

[13] D. Naishlos, "Autovectorization in GCC," GCC Summit, June 2004.

[14] I. Buck, T. Foley, D. Horn, et al., "Brook for GPUs: stream computing on graphics hardware," in *Proceedings of the 31st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '04)*, vol. 23, pp. 777–786, Los Angeles, Calif, USA, August 2004.

[15] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: a language for streaming applications," in *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science, pp. 179–196, Springer, Grenoble, France, April 2002.

[16] P. Mattson, *A programming system for the imagine media processor*, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, Calif, USA, 2001.

[17] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles, "Stream scheduling," in *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pp. 101–106, Austin, Tex, USA, December 2001.

[18] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Ownens, "Communication scheduling," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, pp. 82–92, Cambridge, Mass, USA, November 2000.

[19] W. J. Dally, U. J. Kapasi, B. Khailany, J. H. Ahn, and A. Das, "Stream processors: programmability with efficiency," *ACM Queue*, vol. 2, no. 1, pp. 52–62, 2004.

[20] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the imagine stream architecture," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pp. 14–25, Munich, Germany, June 2004.

[21] W. Wolf, "VLIW architectures for video signal processing," in *Multimedia Hardware Architectures*, S. Panchanathan, F. Sijstermans, and S. I. Sudharsanan, Eds., vol. 3311 of *Proceedings of SPIE*, pp. 52–57, San Jose, Calif, USA, January 1998.

[22] A. Freimann, T. Brune, and P. Pirsch, "Mapping of video decoder software on a VLIW DSP multiprocessor," in *Multimedia Hardware Architectures*, S. Panchanathan, F. Sijstermans, and S. I. Sudharsanan, Eds., vol. 3311 of *Proceedings of SPIE*, pp. 67–78, San Jose, Calif, USA, January 1998.

[23] M. G. Albanesi, M. Ferretti, and A. Dell'Olio, "Effectiveness of a VLIW architecture in a data parallel image application," in *Proceedings of the IEEE International Workshop on Computer Architectures for Machine Perception*, pp. 172–183, New Orleans, La, USA, May 2003.

[24] D. Barretta, W. Fornaciari, M. Sami, and D. Pau, "SIMD extension to VLIW multicluster processors for embedded applications," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '02)*, pp. 523–526, Freiburg, Germany, September 2002.

[25] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz, "The stream virtual machine," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*, pp. 267–277, Antibes Juan-les-Pins, France, September-October 2004.

[26] B. Ramakrishna Rau, "Iterative modulo scheduling," HP Technical Report HPL-94-115, Hewlett-Packard Laboratories, Palo Alto, Calif, USA, November 1995.

[27] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero, "Swing modulo scheduling: a lifetime sensitive approach," in *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 80–87, Boston, Mass, USA, October 1996.

[28] E. Nystrom and A. E. Eichenberger, "Effective cluster assignment for modulo scheduling," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 103–114, Dallas, Tex USA, December 1998.

[29] B. Zwernemann, "An 8 × 8 DCT Implementation on the Motorola DSP56800E," http://www.freescale.com/.

[30] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher," Version 1.1; August 1998.

[31] J. Gummaraju and M. Rosenblum, "Stream programming on general-purpose processors," in *Proceedings of the 38th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '05)*, pp. 343–354, Barcelona, Spain, November 2005.