Compilation of Stream Programs for Multicore Processors that incorporate Scratchpad Memories

Weijia Che, Amrit Panda and Karam S. Chatha,
Faculty of Computer Science and Engineering,
Arizona State University,
Tempe, AZ 85287.
{weijia.che,akpanda,kchatha}@asu.edu

Abstract—The stream processing characteristics of many embedded system applications in multimedia and networking domains have led to the advent of stream based programming formats. Several multicore processors aimed at embedded domains incorporate scratchpad memories (SPM) due to their superior power consumption characteristics. The paper addresses the problem of compiling stream programs on to multicore processors that incorporate SPM. Performance optimization on SPM based processors requires effective schemes for software based management of code and/or data overlay. In the context of our problem instance the code overlay scheme impacts both the stream element to core mapping and memory available for inter-processor communication. The paper presents an integer linear programming (ILP) formulation and heuristic approach that effectively exploit the SPM to maximize the throughput of stream programs when mapped to multicore processors. The experimental results demonstrate the effectiveness of the proposed techniques by compiling StreamIt based benchmark applications on the IBM Cell processor and comparing against existing approach.

I. INTRODUCTION

Increasing performance requirements of many embedded system applications has led to the advent of multicore processor architectures. The Intel IXP series [1], Sun Niagara [2], IBM Cell Broadband Engine (CE) [3], Nvidia GeForce series [4], Intel Larrabee [5], Tilera Tile64 [6] are all instances of multicore processors aimed at embedded domains. These processors achieve high performance in a reasonable power budget by incorporating multiple heterogeneous cores, specialized architectural features, and scratchpad memories (SPM).

The architectural innovations have brought forth the challenge of programming these novel embedded processors. At present a typical design flow does not include a compiler that can effectively parallelize the application and achieve maximal performance. In the absence of such a framework, the designer is required to manually split the application into multiple threads which she assigns to individual cores. Additionally, the designer is also required to include code segments for inter-core communication. The processor vendor does supply traditional compiler tool chains that can compile the threads assigned to individual cores and generate assembly. Thus, performance optimization on multicore embedded processors is a lengthy manual task which leads to inferior implementations.

Recent years have witnessed the emergence of stream programming languages that capture the inherent streaming characteristics of many embedded system applications in the multimedia, network processing and gaming domains. Brook [7], CUDA [8], Baker [9], Sh [10] and StreamIt [11] are some of the stream based programming languages that have been proposed. Stream based formats effectively capture the spatial and temporal parallelism in an application, and therefore are particularly suited for programming on multicore processors. However, the challenges associated with compiling traditional multi-threaded programs on multicore processors (as described above) also hold true for stream languages.

The paper addresses the problem of compiling and optimizing stream programs on embedded multicore processors that incorporate SPM. In particular we consider the compilation of StreamIt programs on the IBM Cell BE. The Cell BE consists of a PowerPC core that acts as a control plane processor and 8 Synergistic Processing Engines (SPEs) that perform the high performance data plane computation. The SPEs are equipped with SPM which brings with them additional programming challenges (described in Section II-C).

Previous research has addressed mapping of synchronous dataflow (SDF) specifications on heterogeneous multicore processors [12]. More recently Chen et al. [13] and Ostler et al. [14] proposed techniques for mapping stream program based specifications on network processors. Liao et al. [15] proposed parallelization schemes for the Brook language on general purpose multicore processors. In contrast to these approaches our research is focussed on embedded multicore processors that have SPMs. There has been recent research that have proposed a dynamic scheduler [16] and compiler optimizations for single threaded code [17] for Cell BE. Our research on the other hand is focussed on static code optimizations and compiling stream programs on Cell BE. Gordon et al. [18] proposed an approach that utilized fusion/fission operators for maximizing the performance of StreamIt programs when compiled for the RAW architecture. In addition to utilizing similar operators, we are also concerned with addressing the trade-off between computation time, code overlay and communication overheads imposed by the SPMs.

The work that comes closest to ours is by Kudlur et al. [19] that also considered the compilation of StreamIt programs on the Cell BE. They proposed an integer linear programming (ILP) formulation for the problem. However, their ILP formulation did not consider memory constraints of the SPM and consequently the code overlay costs associated with a mapping. Further, they also did not model inter-core communication (direct memory access or DMA) overheads in the ILP and assumed that all such overheads could be hidden by computation. However, in several practical instances the SPM code overlay overhead is significant. Further, in many instances the communication overheads cannot be hidden by computation time. Ignoring these overheads leads to inferior designs. Our paper overcomes these limitations and makes the following contributions:

1) An ILP formulation for Compiling Stream programs on SPM equipped Multicore Processors (named as CSMP$_{ilp}$) that models both the code overlay and communication overheads.

2) A fast polynomial time heuristic (named as CSMP$_{heu}$) for the same problem that is able to achieve comparable results as the ILP formulation in a matter of seconds.

We establish the effectiveness of the proposed techniques by mapping StreamIt benchmark programs on Cell BE, and comparing with existing approach [19].
The stream program is described by a SDF $G(F, E)$ where $F$ is the set of filters, and $E$ is the set of FIFOs between filters. Each $f \in F$ is given by a tuple $f(C_f, S_f, \tau_f)$ as described in Table I. Each $e \in E$ is given by $(p_e, inc_e, c_e)$ also described in Table I. In the SDF description we assume that each filter executes only once. Consequently, each FIFO has only one data size ($C_e$) associated with it. It is quite straightforward to transform from the traditional SDF description [20] to our intermediate format.

The objective is to seek a mapping of the SDF on the multicore architecture such that the throughput of the design is maximized subject to the memory constraints. The problem is quite complex as it involves several design trade-offs.

The throughput of a SDF on multicore processor can be optimized by utilizing fusion and fission operators [14]-[19]. For example, consider a linear SDF $G'$ with three filters $A$, $B$, and $C$ with execution times $50\,$ms, $50\,$ms and $200\,$ms, respectively. Further, assume that due to code size constraints only two filters can be mapped on one core. If $G'$ is to be executed on a processor with 3 cores, then fusing (or merging) $A$ and $B$ and executing them sequentially on core 1, and a fission (or replication) of $C$ on cores 2 and 3 maximizes the throughput. However, for large SDFs fusion cannot be applied indiscriminately as cores have a memory restriction beyond which an additional code overlay overhead adversely impacts performance.

Thus, there is a trade-off between the benefit of fusion (as it frees up cores for slower filters) and the resulting overlay overhead. Further, fission cannot be applied in the case of a stateful filter.

In the discussion thus far we have ignored the communication overhead for a FIFO whose producer and consumer filters are mapped to different cores. As the DMAs can be launched in a non-blocking manner, it is possible to amortize the communication overhead with filter execution. However, this scheme (also known as double buffering) requires more memory space. The FIFO is assigned two locations on the producer and consumer core’s memory, respectively. While the producer and consumer read data from one location in their respective FIFO buffers, a DMA operation transfers data between the other two FIFO buffers. As the FIFO is assigned on the same memory as the code, there is a trade-off between the memory usage of the two, and the resulting performance.

Finally, in the above discussion we assumed that the communication overhead can be effectively hidden by double buffering. However, it may not be the case if the communication overhead itself is too large. In addition to considering overlay overheads, the fusion/fission operators and mappings must also take into account the resulting communication overheads.

In the following sections we describe an ILP formulation and heuristic approach that are both able to effectively address the various design trade-offs and generate high quality solutions.

---

**A. IBM Cell Broadband Engine**

In this paper we target the IBM Cell Broadband Engine (BE) as an instance of multicore processor architectures aimed at high performance embedded system applications. Figure 1 depicts the architecture of the Cell BE [3]. The processor consists of a single PowerPC based PPE, and 8 synergistic processing elements (SPE) operating at 3.2 GHz. The dual-threaded PPE runs the conventional operating system and is responsible for launching threads on the SPE. The SPEs support 4-way SIMD and are the primary workhorses for the processor. Each SPE is supported by 256KB scratch pad memory that acts as the local store. The element interconnect bus connects the PPE, SPEs and the memory controllers, and provides a cumulative bandwidth of over 204.8 GBps. Direct memory access or DMA (which can be launched by either the PPE or SPEs) is the primary means for communicating between the local store of two SPEs or SPE local store and the main memory. The non-blocking nature of the DMA permits the amortization of communication overhead by concurrent computation. Up to 16 independent DMAs can be launched simultaneously by each initiating core (PPE or SPE).

**B. StreamIt**

We utilize StreamIt [11] as the programming language for specifying the streaming applications. StreamIt which was developed at MIT implements the synchronous dataflow (SDF) [20] model of computation. A filter is the basic unit of computation in StreamIt. Each filter communicates with other filters through FIFOs. A filter can execute as long as it has sufficient data items on its input FIFOs and sufficient space on the output FIFOs to store the data. Thus, a filter executes in a blocking read/write manner. On each execution the number of data items produced/consumed on FIFOs is constant. StreamIt also provides three constructs for program composition, namely pipeline, split/join and feedback loop.

**C. Problem description**

The inputs to the problem consist of the architectural description of the target multicore processor and SDF based intermediate format that captures the stream program. The cores in the architecture are described by a set $P$ where each $p \in P$ is given by a tuple $p(C_p, L_p, T_{init}, D_{init}, T_{slope})$ as described in Table I. During the characterization of the Cell BE it was found that the DMA overhead is constant at $2.1\,\mu s = T_{init}$ below a block size of $1\,KB = D_{init}$. Beyond $1\,KB$ the DMA transfer overhead was found to increase at $0.075\,\mu s/K\,B = T_{slope}$.

The CELL BE also supports signals and mail boxes for inter-SPE communication. However, these are primarily useful for synchronization or for very small data items. As we are more concerned about large data items we focus on DMA.

---

**TABLE I**

<table>
<thead>
<tr>
<th>Arch.</th>
<th>$C_p$</th>
<th>Local memory size of the processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_p$</td>
<td>1 (true) if overlay overhead exists for the local mem.</td>
<td></td>
</tr>
<tr>
<td>$T_{init}$</td>
<td>Lowest DMA transfer overhead</td>
<td></td>
</tr>
<tr>
<td>$D_{init}$</td>
<td>Largest DMA size that can be transferred with $T_{init}$</td>
<td></td>
</tr>
<tr>
<td>$T_{slope}$</td>
<td>Rate of increase of DMA overhead beyond $D_{init}$</td>
<td></td>
</tr>
<tr>
<td>SDF</td>
<td>$C_f$</td>
<td>Size of code and local data for filter $f$</td>
</tr>
<tr>
<td>$S_f$</td>
<td>1 (true) if $f$ is non-fissable (stateful)</td>
<td></td>
</tr>
<tr>
<td>$\tau_f$</td>
<td>Running time of filter $f$</td>
<td></td>
</tr>
<tr>
<td>$p_e$</td>
<td>Producer of FIFO $e$</td>
<td></td>
</tr>
<tr>
<td>$inc_e$</td>
<td>Consumer of FIFO $e$</td>
<td></td>
</tr>
<tr>
<td>$c_e$</td>
<td>Data produced/to consumed from $e$</td>
<td></td>
</tr>
</tbody>
</table>
III. INTEGER LINEAR PROGRAMMING APPROACH

We describe an ILP approach (called CSMP\textsubscript{ulp}) for compiling stream programs on multicore processors incorporating with SPMs. In our ILP the fission and fusion operators are implemented by first assigning the filters to batches, and then the batches to processors. In the mapping, each filter must be assigned to exactly one batch and each processor must be assigned one batch to execute. Figure 2.A provides an example of the mapping with 6 filters and 3 processors and Figure 2.B sketches the steady-state execution. Ostler el al. [14] proved that such a batching strategy can generate optimum solutions for stream programs. The base and derived variables of the ILP are described in Table II.

A. Constraints

The constraints of our ILP are described below. As some of the constraints are identical to the ILP by Ostler et al. [14], the corresponding equations are not included.

File read and write: The source and sink filters are assumed to be file read and write operations, respectively that can only execute on the constraints are identical to the ILP by Ostler et al. [14], the corresponding equations are not included.

Filter to batch assignment: Each filter is mapped to one and only one batch [14].

Batch to processor assignment: If batch \( b \) has \( n \) copies, then exactly \( n \) processors must be assigned to execute \( b \) [14].

Processor utilization: Each processor must be assigned exactly one batch to execute [14].

Batch utilization: A batch needs to be mapped to some processor to execute only when there is at least one filter being assigned to it [14]. Nonfissable filter: If a batch consists of a non-fissable filter then it has only one copy.

\[
a_{ij} = 1, a_{j|f|} = 1, b_{ij} = 1, S_1 = 1, n_{11} = 1 (1)
\]

\[
\begin{align*}
S_b \geq S_f &+ n_{fb} \times S_f \quad \forall f \in F, b \in B \quad (2) \\
S_b &\leq \sum_{f \in F} a_{fb} \times S_f \\
n_{fb} \geq S_b &/ \text{nonfissable batch has 1 copy} / \\
\end{align*}
\]

MAX\_VAL is some large value and \( S_b \) indicates whether batch \( b \) is nonfissable.

Iteration assignment: Each filter runs at some iteration number. If the producer and consumer of an edge \( e \) are being assigned to different batches, then the producer runs at least \( 2 \) iteration numbers higher than the consumer.

\[
\forall f \in F: \sum_{n \in N} i_{fn} = 1, \forall e \in E: I_e \geq 2 \times K_e (3)
\]

\textbf{Table II: Base and Derived Variables}

<table>
<thead>
<tr>
<th>Var</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{fb} )</td>
<td>INTEGER</td>
<td>filter ( f ) belongs to batch ( b )</td>
</tr>
<tr>
<td>( b_{op} )</td>
<td>INTEGER</td>
<td>batch ( b ) is assigned to processor ( p )</td>
</tr>
<tr>
<td>( n_{fb} )</td>
<td>INTEGER</td>
<td>number of replicated copies of batch ( b )</td>
</tr>
<tr>
<td>( i_{f} )</td>
<td>INTEGER</td>
<td>iteration number of filter ( f )</td>
</tr>
<tr>
<td>( p_f )</td>
<td>REAL</td>
<td>1 if filter ( f ) always in SPM</td>
</tr>
</tbody>
</table>

\textbf{Buffer Usage:} The buffer usage of batch \( b \) is calculated by its incoming \((I_e + C_e)\), outgoing \((2 \times C_e)\) and internal \((I_e + 1) \times C_e\).

\[
C_b(bu) := \sum_{e \in E} IC_{eb} + \sum_{e \in E} 2 \times PE_{eb} + \sum_{e \in E} (IM_{eb} + IM_{eb}) \times C_e (4)
\]

Code Overlay: The code overlay overhead is given by the fetch time for all the filters that are not present in the SPM.

\[
\forall b \in B : \gamma_{b}(overlay) := \sum_{f \in F} (a_{fb} - X_{fb}) \times (T_{init} + T_{slope} \times C_f) (5)
\]

Processor Memory: The sum of the buffer usage and the region for code overlay must be less than the processor local memory size.

\[
\forall b \in B, p \in P : C_{bp} \leq b_{op} \times \text{MAX\_VAL} (6)
\]

\[
C_{bp} \geq C_b(bu) + C_{bp}(code) + (b_{op} - 1) \times \text{MAX\_VAL} (7)
\]

B. Objective function

The execution time of a batch is given by the maximum of its computation (which includes code overlay overheads) and communication time. The effective execution time of a batch equals the execution time of that batch divided by the number of its copies. If a batch has multiple copies, we also introduce a fission overhead for additional split/join nodes and potentially more peek operations (captured by \( c \)).

Computation time: The computation time of batch \( b \) is given by the sum of the computation time of all the filters being assigned to it plus its overlay overhead.

\[
\forall b \in B : \tau_{b} := \sum_{f \in F} a_{fb} \times i_{f} + \gamma_{b}(overlay) (7)
\]

Communication time: The communication time of batch \( b \) is given by the time to fetch all its input data to the local memory in the
steady-state execution.

\[ \forall b \in B : \tau'_b := \sum_{e \in E} C'_{eb} \times (T_{init} + T_{slope} + C'_e) \]  

(8)

**Execution time:** The execution time of a batch is the maximum of its computation and communication times.

\[ \forall b \in B : \tau_b \geq \tau'_b, \quad \tau_b \geq \tau'_b \]  

(9)

**Effective execution time:** Given by

\[ \forall b \in B : \Gamma_b := \sum_{n=1}^{[P]} \frac{1}{n} \tau_{bn} + \sum_{f \in F} Y_{fb} \times (e + \text{split}_\text{join}_\text{work}) \]  

where, \( \tau_{bn} \) is given by

\[ \forall b, n \in [1, \lfloor P \rfloor] : \tau_{bn} \leq n_{bn} \times \text{MAX}_\text{VAL} \]  

\[ \tau_{bn} \geq \tau_b + (n_{bn} - 1) \times \text{MAX}_\text{VAL} \]  

\[ \tau_{bn} \geq 0 \]  

(11)

The objective function is to minimize effective execution time over all batches and thus, maximize the throughput.

\[ \forall b \in B : \Gamma \geq \Gamma_b \quad \text{minimize}(\Gamma) \]  

(12)

### IV. Heuristic Approach

In this section, we present a heuristic approach (called CSMP\( \text{heu} \)) that can be utilized for compiling stream programs on SPM enhanced multi-core processors. The heuristic involves iterative application of fusion/fission operators, and estimation of the performance of the resulting SDF. The performance estimation considers both overlay costs and communication costs. The overlay costs take into account the trade-off between the buffer requirement for communication and code memory. Although the application of fusion/fission operators is similar to Ostler et al. [14], our approach can address the design complexity introduced by the SPM.

The main routine of our approach is shown in Figure 3. The function \( \text{initialize}() \) assigns each filter \( f \in G \) to a distinct batch \( b \in B \). The non-fissable batches that include a non-fissable filter are denoted by \( B_{nf} \) (\( B_{nf} = B/B_{nf} \)). \( \text{initialize}_\text{iteration}() \) assigns the iteration number of every filter. As each batch is assigned to a different processor the difference between the iteration numbers of producer and consumer filters is set as 2. As explained earlier this enables overlapping of DMA transfers with computation. We next calculate \( K = C(B_{nf}, f, B, P) \) (given below) which denotes the number of batches that non-fissable filters should be fused into. \( K \) is given by the product of number of cores and the summation of run time of the non-fissable batches over the summation of the run time of all batches.

\[ K := \frac{\sum_{f \in B_{nf}} \tau_f}{\sum_{f \in B} \tau_f} \times [P] \]  

(13)

The function \( \text{fission}_\text{number}(B_{nf}, f, K) \) fuses the input set of batches \( (B_{nf}) \) into \( K \) distinct batches. The fusion operation considers all pairs of batches and fuses the pair which has the lowest total cost after the merge. Next we fuse the batches using function \( \text{fission}_\text{number}(B, |P|) \). Thus, the total number of batches are now equal to the number of cores. Let \( C \) denote the effective execution time of this design.

We next iteratively (within the for loop) explore different design alternatives that can improve upon the initial solution. In each iteration we first generate a solution by fusing fissionable batches. For example in the first iteration we fuse and reduce the number of fissionable batches by one. We then apply the fission operator that iterative replicates the slowest batches until the total number of batches are the same as the number of cores. We save the solution \( (B_s) \) that gives the lowest cost. At the end of the for loop we return the best solution.

The computational complexity is determined by the for loop. The computational complexity of the fuse operation is \( O(n^2) \) as it considers all pairs of batches. The complexity of \( \text{fission}_\text{number}() \) is \( O(n^3) \), and consequently the complexity of the overall runtime is \( O(n^4) \). In the following we elaborate upon the overlay scheme and cost function calculations utilized by our approach.

#### A. Overlay Scheme

We utilized a greedy overlay scheme in the interest of efficiency. We first calculate the buffer usage of the batch \( b \), and derive the memory available for code (= \( C_{\text{code}} \)). If \( C_{\text{code}} \) is able to accommodate all the filters in batch \( b \) then code overlay is not required. Otherwise, we need to determine an overlay scheme and estimate the resulting overhead. We first assign as many filters as possible into the overlay code and utilize the available memory \( C_{\text{overlay}} \) to overlay the remaining filters. We sort the remaining filters of \( b \) in decreasing order of their iteration number and assign them to segments \( (S_i) \) as long as the segment size does not exceed the \( C_{\text{overlay}} \). The overlay overhead is given by:

\[ \tau_{\text{overlay}}(b) = \sum_{S_i \in b} \tau(S_i) \]  

\[ \tau(S_i) = \begin{cases} T_{\text{init}} & \text{if } |S_i| \leq D_{\text{init}} \\ T_{\text{init}} + (|S_i| - D_{\text{init}}) \times T_{\text{slope}} & \text{Otherwise} \end{cases} \]  

(14)

#### B. Cost functions

In the following paragraphs we detail the calculation of buffer usage, the computation cost, and the communication cost.

1) **Buffer usage:** The buffer usage of a batch \( b \) is given by the memory required for storing all the incoming, outgoing and internal data of \( b \). Therefore, it is given by:

\[ \text{buf}(b) = \sum_{e \in b} \text{buf}(e) \]  

\[ \text{buf}(e) = \begin{cases} (I_e + 1) \times C_e & \text{if } f_{pe}, f_{ce} \in b \\ 2 \times C_e & \text{if } f_{pe} \in b, f_{ce} \notin b \\ I_e \times C_e & \text{if } f_{pe} \notin b, f_{ce} \in b \end{cases} \]  

(15)

2) **Computation cost:** The computation cost of batch \( b \) is given by the sum of the computation time of all the filters being assigned to it plus its overlay overhead.

\[ \tau_{\text{comp}}(b) := \tau_{\text{overlay}}(b) + \sum_{f \in b} \tau_f \]  

(16)
3) Communication cost: The communication cost of batch $b$ is given by the time for $b$ to fetch all its data in the steady-state execution.

$$
\tau_{comm}(b) = \sum_{e \in \text{fp}_b \& \text{fp}_e} \tau_{comm}(e)
$$

(17)

$$
\tau_{comm}(e) = \begin{cases} 
T_{init} & \text{if } C_e \leq D_{init} \\
T_{init} + (C_e - D_{init}) \times T_{slope} & \text{otherwise}
\end{cases}
$$

4) Overall cost of solution: The cost of a batch is given by $\tau(b) = \max(\tau_{comm}, \tau_{comp})$. The effective cost of a batch is given by $\tau_{eff}(b) = N/\tau(b)$ where $N$ is the number of copies of $b$. Finally, the overall cost of the application is given by the largest effective execution time over all batches.

V. EXPERIMENTAL RESULTS

In this section we present experimental results that evaluate CSMP$_{ilp}$ and CSMP$_{heu}$, and compare them with the SGMS approach [19] which is also an ILP. We utilized StreamIt benchmarks that are delivered with the 2.1.1 version of the compiler. We targeted the benchmark applications for Sony PS3 platform that includes the Cell BE. Due to hardware constraints of the platform only 6 SPEs and the PPE are available to the application developer. We utilized the StreamIt compiler to generate C implementations which were then passed to the respective gcc compilers to obtain profile information on the SPE and PPE.

A. Comparisons with 256KB SPE memory

We first ran the three techniques with 256KB SPM which is the size of the SPE local store. Figure 4 presents the computation and communication costs of the SGMS solutions. The $y$-axis in the figure stands for the steady-state execution time of each benchmark normalized to its lower bound. The lower bound is calculated by the total execution time of all the filters in that benchmark over the number of processors. As is illustrated in Figure 4, for 4 benchmarks the SGMS solutions have their communication costs overwhelming the computation costs. In the extreme case, for the bitonic sort benchmark, the communication cost is more than 10x over the computation cost. Figure 5 presents the computation and communication costs of our CSMP$_{ilp}$ and CSMP$_{heu}$ algorithms. In our ILP solutions, there is only one benchmark with its communication cost larger than the computation cost. The ratio, however, is only 1.1x, a much smaller number comparing to 10x as in the SGMS solutions. In the solutions generated by CSMP$_{heu}$, the communication costs are hidden by the computation costs for all benchmarks. Figure 6, compares the overall performance of the SGMS, CSMP$_{ilp}$ and CSMP$_{heu}$. As is observed from the figure, CSMP$_{ilp}$ always performs better than the SGMS. CSMP$_{heu}$ also outperforms the SGMS on average. The above results show that our techniques are able to effectively trade-off the computation and communication costs to balance the overall performance.

In Table III we compare the run time of the SGMS, CSMP$_{ilp}$ and CSMP$_{heu}$. On average, the SGMS took 209.5 seconds to finish and CSMP$_{ilp}$ took 28387 seconds. The numbers indicate the SGMS is over 100 times faster than CSMP$_{ilp}$. The primary reason is that, SGMS tries to do load balancing solely based on the steady-state execution time of each filter. CSMP$_{ilp}$ takes the whole graph as input and does load balancing based on the filter execution time, the data communication, the current schedule, the buffer usage, and the overlay scheme. On the other hand, CSMP$_{heu}$ is very fast even though it considers all the design trade-offs. 7 out of the 12 benchmarks terminate within 1 second. The serpentfull benchmark requires the longest run time as it has 120 filters and 128 edges.

In summary both our techniques out perform SGMS. In particular, our heuristic approach is able to generate high quality solutions (27.8% slower than our ILP on average, 44.5% faster than SGMS on average) in very short run times.

B. Comparisons with 16KB SPE memory

The solutions for the previous experiment did not require code overlays due to the large size of the SPM. We conducted a second set of experiments with SPM size constrained to 16KB. We timed out CSMP$_{ilp}$ after 9 hours and utilized the solution if it was feasible. CSMP$_{ilp}$ was unable to generate solutions for 4 benchmarks (namely mpeg2, serpentfull, tde and vocoder). Table IV lists the maximum SPE communication buffer usage for SGMS, CSMP$_{ilp}$ and CSMP$_{heu}$. SGMS was unable to find solutions in four instances (shown in bold) as the communication buffer requirement violated the SPM memory constraint. The overall performance of the SGMS, CSMP$_{ilp}$ and CSMP$_{heu}$ are presented in Figure 7. Some of the CSMP$_{ilp}$ data points are missing as it timed out. In the case of SGMS the data points that are missing (dct, fft, mpeg2, tde and vocoder) indicate infeasible solutions. CSMP$_{heu}$ was able to generate feasible solutions for all benchmarks. Except for two benchmarks (channel vocoder and serpentfull) our heuristic is able to out perform SGMS.

In Table IV we compare the run time of the SGMS, CSMP$_{ilp}$ & CSMP$_{heu}$ (seconds).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SGMS</th>
<th>CSMP$_{ilp}$</th>
<th>CSMP$_{heu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>beamformer</td>
<td>1261</td>
<td>4172</td>
<td>0.67</td>
</tr>
<tr>
<td>bitonic sort</td>
<td>80</td>
<td>4124</td>
<td>0.46</td>
</tr>
<tr>
<td>channel vocoder</td>
<td>59</td>
<td>26319</td>
<td>1.38</td>
</tr>
<tr>
<td>dct</td>
<td>1541</td>
<td>91764</td>
<td>0.51</td>
</tr>
<tr>
<td>des</td>
<td>165</td>
<td>33083</td>
<td>1.19</td>
</tr>
<tr>
<td>fft</td>
<td>29</td>
<td>3328</td>
<td>0.08</td>
</tr>
<tr>
<td>filterbank</td>
<td>236</td>
<td>42297</td>
<td>10.86</td>
</tr>
<tr>
<td>fm</td>
<td>59</td>
<td>38541</td>
<td>0.51</td>
</tr>
<tr>
<td>mpeg2</td>
<td>101</td>
<td>16364</td>
<td>0.11</td>
</tr>
<tr>
<td>serpentfull</td>
<td>56</td>
<td>23853</td>
<td>55.83</td>
</tr>
<tr>
<td>tde</td>
<td>116</td>
<td>34412</td>
<td>0.15</td>
</tr>
<tr>
<td>vocoder</td>
<td>60</td>
<td>22387</td>
<td>5.62</td>
</tr>
</tbody>
</table>

SGMS and CSMP$_{ilp}$ ran on our server with Quad-Core Intel(R) Xeon(TM) CPU at 2.8GHz and the CSMP$_{heu}$ ran on our PC with Intel(R) Core(TM)2 Quad CPU at 2.4GHz.

TABLE III

MAXIMUM SPE COMMUNICATION BUFFER USAGE OF SGMS, CSMP$_{ilp}$ & CSMP$_{heu}$ (BYTES).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SGMS</th>
<th>CSMP$_{ilp}$</th>
<th>CSMP$_{heu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>beamformer</td>
<td>1004</td>
<td>1156</td>
<td>1026</td>
</tr>
<tr>
<td>bitonic sort</td>
<td>400</td>
<td>288</td>
<td>564</td>
</tr>
<tr>
<td>channel vocoder</td>
<td>10216</td>
<td>11212</td>
<td>12400</td>
</tr>
<tr>
<td>dct</td>
<td>196608</td>
<td>10240</td>
<td>8192</td>
</tr>
<tr>
<td>des</td>
<td>8320</td>
<td>13708</td>
<td>12604</td>
</tr>
<tr>
<td>fft</td>
<td>24576</td>
<td>14336</td>
<td>14436</td>
</tr>
<tr>
<td>filterbank</td>
<td>2588</td>
<td>3472</td>
<td>1920</td>
</tr>
<tr>
<td>fm</td>
<td>204</td>
<td>328</td>
<td>248</td>
</tr>
<tr>
<td>mpeg2</td>
<td>24742</td>
<td>1126</td>
<td>NA</td>
</tr>
<tr>
<td>serpentfull</td>
<td>10204</td>
<td>NA</td>
<td>5836</td>
</tr>
<tr>
<td>tde</td>
<td>122880</td>
<td>NA</td>
<td>0</td>
</tr>
<tr>
<td>vocoder</td>
<td>2636</td>
<td>NA</td>
<td>5756</td>
</tr>
</tbody>
</table>

The bold faced values indicate that SPM capacity is exceeded. CSMP$_{heu}$ maps all filters to PPE for tde benchmark. Hence, the SPE buffer usage is 0.

In summary both our techniques out perform SGMS. In particular, our heuristic approach is able to generate high quality solutions (27.8% slower than our ILP on average, 44.5% faster than SGMS on average) in very short run times.
VI. Conclusion

We presented an ILP formulation and heuristic approach for compiling stream programs on embedded multicore processors that incorporate SPMs. We evaluated the approaches by compiling StreamIt programs on the Cell BE processor, and comparing with an existing technique. The experimental results showed that our approaches are able to effectively balance the computation (including overlay) and communication overheads when mapping stream programs on multicore processors. Further, our heuristic approach is able to generate high quality solutions even under tighter SPM capacity constraints while the existing technique produces infeasible solutions.

Acknowledgement

The research presented in the paper was supported in part by grants from Science Foundation Arizona (SFAz), Stardust Foundation and National Science Foundation (CCF - 0903513).

References