# The Hyperprocessor: A Template System-on-Chip Architecture for Embedded Multimedia Applications[*]

Faraydon Karim, Alain Mellan,
Bernd Stramm, Anh Nguyen
STMicroelectronics
4690 Executive Drive
San Diego, CA 92121
{faraydon.karim,alain.mellan,
bernd.stramm,anh-q.nguyen}@st.com

Tarek Abdelrahman, Utku Aydonat
Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{tsa,uaydonat}@eecg.toronto.edu

## Categories and Subject Descriptors

C.1.3 [**Computer System Organization**]: Processor Architectures—*Heterogeneous (hybrid) systems, Adaptable architectures*; D.1 [**Software**]: Programming techniques, Sequential Programming, Concurrent Programming

## General Terms

Design, Performance

## Keywords

Hyperprocessor, superscalar, multi-processor, heterogeneous, compiler

## ABSTRACT

We describe and evaluate a template architecture for SoC systems intended for multimedia applications. The architecture is a 2-level hierarchy that consists at the bottom level of several processing units (PUs), controlled at the top level by a control processor. The main characteristic of our architecture is that it exploits in hardware parallelism among tasks executing on different PUs in the same way a superscalar processor exploits instruction level parallelism. This hardware support for task-level parallelism gives rise to a natural programming model that relieves programmers from explicitly synchronizing tasks and communicating data. We describe a number of code transformations to improve performance that are based on well-known compiler analyses, and thus can be incorporated into a compiler for this architecture. We use simulation, two realistic multimedia applications and the proposed code transformations to explore the performance vs. resources trade-off.

[*]This work was supported by a research grant from STMicroelectronics.

## 1. INTRODUCTION

A solution paradigm that has emerged in the embedded systems market over the last few years is that of a programmable System-on-a-Chip (SoC): an integrated design that incorporates possibly multiple programmable cores and various custom or semi-custom blocks and memories into a single chip. This paradigm allows the reuse of pre-designed cores (commonly referred to as intellectual property, or IP), thus amortizing the design cost of a core over many system generations.

In this context, we designed a novel architecture to simplify integration of heterogeneous IP for multimedia and streaming applications. The *Hyperprocessor* is template architecture that features multiple processing units and a top level controller that automatically exploits parallelism among coarse-grain units of computation, or *tasks*, using well-deve-loped superscalar processor technology. The Hyperprocessor supports a programming model, which, similar to that of sequential programming, does not require programmers to specify synchronization and/or data communication. Compiler technology to improve the performance of Hyperprocessor programs is developed at the same time as the architectural design is completed.

In this paper, we describe the Hyperprocessor, its architecture and its programming model. In particular, we explore the benefits of the architecture, and use two realistic multimedia applications to show that minimal effort is needed in porting these applications to the Hyperprocessor. We further describe a set of code transformations for improving performance of programs on the Hyperprocessor. These transformations are based on standard compiler analyses, and hence, can be incorporated into compilers for the Hyperprocessor. We use simulation, the two applications, and the proposed code transformations to explore the performance vs. resources trade-off of different instances of the architecture. In particular, we find that adding more processors to the architecture results in scaling performance, and that there is negligable contention over resources in the top level control processor. These results lead us to believe that the Hyperprocessor is a viable architecture for SoC solutions for multimedia applications.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Hyperprocessor. Section 3 describes the two applications we used as case studies, and the transformations applied to their code to enhance performance. Section 4 presents our experimental evaluation of the Hyperprocessor. Section 5 describes related work. Finally, Section 6 given some concluding remarks and directions for future work.

## 2. THE HYPERPROCESSOR

### 2.1 Architecture

The Hyperprocessor is a 2-level hierarchical architecture that at the lower level consists of multiple *processing units* (PUs). A PU can be a full-fledged processor core (superscalar, VLIW, etc), a DSP, a block of FPGA, or some custom hardware. The upper level consists of a *control processor* (CP), a *task dispatcher* (TD), and a *universal register file* (URF). A dedicated interconnect links the PUs to the URF and to memory. A block diagram of the Hyperprocessor is shown in Fig. 1(a). The architecture bears considerable similarity to an abstract microprocessor architecture in Fig. 1(b).

The novelty of the Hyperprocessor architecture stems from the fact that the upper level of the hierarchy supports out-of-order, speculative and superscalar execution of coarse-grain units of computation, which we refer to as *tasks*. It does so using the same techniques used in today's superscalar processors, such as register renaming and out-of-order execution, to exploit parallelism among instructions. This leverages existing superscalar technology to exploit task-level parallelism across PUs in addition to possible instruction-level parallelism within a PU.

The CP fetches and decodes *task instructions*, each of which specifies a task to execute. A task instruction also specifies the inputs and outputs of the task as registers in the URF. Dependences among task instructions are detected in the same way that dependences among instructions are detected in a superscalar processor: by means of the source and sink registers in the URF. The CP renames URF registers as necessary to break false dependences among task instructions. Decoded task instructions are then issued to the TD unit. Based on dynamic dependences, tasks can be issued out-of-order, and may also complete and commit their outputs out-of-order.

Task instructions are enqueued in the TD unit, in a similar way instructions are enqueued in the instruction queue of a superscalar PU. When the operands of a task instruction are ready, the task instruction is dispatched using a *scheduling strategy* to the PUs. The simplest such strategy dispatches instructions to PUs in a round-robin fashion. However, more dynamic strategies can also be used.

The Hyperprocessor is a template architecture. It does not specify the form of the interconnect among the PUs. Several implementations are possible, including buses, crossbars, and multi-stage interconnects (for certain configurations, we have worked on a dedicated multi-stage type interconnect called Octagon [10]). In addition, since inter-task dependences are enforced by the CP, and since data communication is primarily accomplished through the URF, there is no need to assume a particular memory architecture. The PUs may share a single memory, or may each have its own private memory, or any combination of the two, depending on the application.

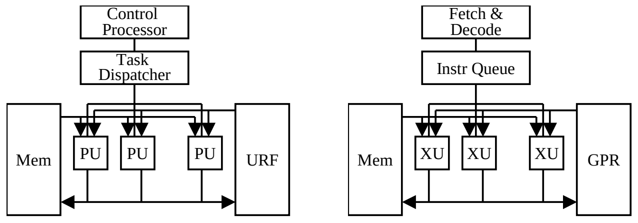
### 2.2 Programming Model

The hardware features of the Hyperprocessor give rise to a natural programming model that is very similar to sequential programming. The Hyperprocessor programming model is layered. The bottom layer comprises the *task bodies*, or simply the *tasks*. Each task implements a given functionality and has defined inputs and outputs. A task can be a sequential `C` program, a block of assembly code executing on a programmable PU such as a processor or DSP core, or a predefined functionality of a non-programmable PU such as a hardware block.

The top level of the Hyperprocessor programming model is a single task-program that executes on the CP of the Hyperprocessor. It is a sequential program that specifies task instructions, and is expressed, for example, in a `C`-like language called Sarek. The language replaces function calls with task calls, and adds explicit direction indications (*in* or *out*) for function arguments.

Fig. 2 and 3 illustrate the Hyperprocessor programming model. The task `Add` shown in Fig. 2 is expressed as a `C` function that computes the sum of two integers. The function has no formal arguments. Instead, it communicates with the Sarek program through an API, obtaining input data with a *readArg* call, and writing results using an analogous *writeArg* call. Both calls specify the input and/or the output using a positional argument. Thus `readArg(1)` reads the second input to the function, while `writeArg(0)` writes the first output of the function. The task also returns a condition code that is written to a *condition register* in the CP, and may be used in a Sarek program to make control decisions, such as jumps or branches.

The main part of the corresponding Sarek program for the example is shown in Fig. 3. It makes four calls to the task `Add`. In each call, the variable names of the inputs and outputs of each task are specified. In addition, a direction indicator (`in` or `out`) are also specified for each variable. The second instance of the task `Add` must wait for the first instance to complete because of the true dependence caused by the use of the variable `totwidth`. However, the third and fourth instances of `Add` may proceed out-of-order with the first two even though they respectively write and read to `totwidth`, and also because there is no dependency with the conditional call to `Div`. The hardware will automatically rename the register holding `totwidth` for these two tasks to eliminate the



(a) Macro-architecture (b) Micro-architecture

**Figure 1: Hyperprocessor high-level architecture analogy.**

```
int Add() {
    int n1 = readArg(0);
    int n2 = readArg(1);

    writeArg(0, n1 + n2);

    return (n1+n2) != 0 ;
}
```

**Figure 2: An example task body.**

```
do {
    ...
    notzero = Add(in width1, in width2,
                  out totwidth);
    notzero = Add(in width3, in totwidth,
                  out totwidth);
    if (notzero) {
        Div(in area1, in totwidth, out length1);
    }
    notzero = Add(in width4, in width5,
                  out totwidth)
    notzero = Add(in width6, in totwidth,
                  out totwidth);
    if (notzero) {
        Div(in area2, in totwidth, out length2);
    }
    ...
    notfinished = NotDone(in index);
} while(notfinished);
```

**Figure 3: The Sarek code of the example.**

```
Do1Top:
        ...
        task Add, CR1, R5:r,R3:r,R3:w
        if false (CR1 & 0x7fffffff) jmpa If2False
If2True:
        task Div, R5:r,R3:r,R4:w
If2False:
        ...
If2End:
        task NotDone, CR2, R2:r
        if true (CR2 & 0x7fffffff)  jmpa Do1Top
```

**Figure 4: Sample HASM code for Sarek fragment of Fig. 3.**

false output and anti dependence that exist.

Sarek has the usual control flow constructs found in most languages, with a syntax similar to the C-language. Sarek has only two data types: *control* variables and *data* variables. Control variables store the return values of task calls, and are used to decide flow of control in conditionals and loops. Sarek allows bitwise logical expressions on control variables. Data variables provide input and output arguments for task calls, as illustrated in the example above.

Semantically, data variables that are output from tasks in a PU are available when the task writes them using the `writeArg` call. By contrast, control variables are available to the upper layer only when a task has completed execution. Consequently, the first conditional `if (notzero) ...` in Fig. 3 can be evaluated only after the preceding task `Add` has completed, even though the input argument `totwidth` for the following task `Div` is available earlier.

Sarek is compiled to generate an intermediate representation of the program similar to assembly, which we call HyperAssembler (HASM). The HASM code fragment in Fig. 4 corresponds to the Sarek code in Fig. 3. In this HASM, the Sarek control variables are stored in *control register*, denoted CR$x$. Sarek data variables are stored in Universal registers, denoted R$x$. For Universal registers, the usage of the register is also given, as `:r` for inputs, and `:w` for outputs, which is used for dependency analysis by the hardware.

## 2.3 Benefits

In our view, the combination of architecture and programming model of the Hyperprocessor give rise to a number of advantages for SoC designs:

- *Reduced software complexity.* The programming model of the Hyperprocessor is close to that of sequential programming, and alleviates the need for explicit parallel programming
- *Automatic extraction of the parallelism.* Similar to instruction-Level parallelism, speedup can be achieved through register renaming and out-of-order execution. For dependencies like Write-After-Write (WAW) and Write-After-Read (WAR), the CP allocates new registers, allowing the tasks to run in parallel on separate PUs.
- *Multiple levels of parallelism.* The Hyperprocessor combines task-level parallelism and instruction-level parallelism. Task-level parallelism is potentially higher than the instruction-level parallelism that can be extracted from sequential code [18].
- *Separation of communication and synchronization from computations.* Synchronization and communication are often significant contributors to the complexity and cost of an embedded system. Both can lead to contention on interconnects, increased latency, and power consumption. In addition, software development is complicated by the need to insert synchronization and communication primitives.
- *Scheduling policy is independent from source code.* Our layered approach fits in the layered model advocated by Thomas and Paul [12], where each layer (application, schedulers, resources) can be tuned independently of the others to attain an optimal cost/performance ratio.
- *Efficient communication.* Tasks may communicate through the URF instead of relying on shared memory.

## 3. THE APPLICATIONS

In this section we present two case studies of porting realistic multimedia applications to the Hyperprocessor. For each case, we describe the application, then the code transformations that were necessary to port the code and obtain good performance.

The code transformations used to improve performance described below builds on a number of well-known compiler analyses and optimizations, including data flow analysis [11], array privatization [16], code hoisting [11] and loop unrolling [11]. Hence, they are easily incorporated into Sarek and C compilers.

## 3.1 MAD

MAD is an MPEG decoder that translates MPEG files into 16-bit PCM output [5]. We use a stripped-down version of the code, which does not include multithreading, but retains the functionalities and code structure of the original application.

The input to MAD is a byte stream that represents a sequence of audio frames. Each frame consists of a frame header[1] and frame data. The frame header contains configuration information such as audio layer type, channel mode, sampling frequency, stream bit rate, and the location of the frame's main data in the input stream. Since frames may be of different sizes, a frame header also contains the size of its corresponding frame.

The main data structure in MAD is a C structure called `mad_decoder`. It contains some global variables and three other C structures: `mad_stream`, `mad_frame`, and `mad_synth`. The `mad_stream` structure stores the start and end addresses

[1]The header consists of two parts; a header part and a "side-info" part. For simplicity, we shall refer to both as the frame header.

```
allocate data structures
map input file to memory
while(!eof) {
    decode header
    decode data into mad_frame
    synthesis pcm output into mad_synth
    send to output
}
unmap input file
deallocate data structures
```

**Figure 5: The main steps of MAD.**

of the input stream in memory, a pointer to the start of the current frame being decoded, a pointer to the next frame to be decoded, and buffers used for decoding a frame. The `mad_frame` and `mad_synth` structures hold buffers for the decoded output and the PCM output for a frame, respectively. Thus, most of the pointers and buffers within `mad_stream`, as well as within the `mad_frame` and `mad_synth` structures are re-used for the decoding of each frame.

Fig. 5 illustrates a break down of MAD's functionalities. The various data structures used by the program are first allocated and initialized. The file containing the input stream is then mapped to memory, and the frames are decoded one at a time until end-of-file is reached. For each frame, the decoded output is copied to `mad_frame`. The PCM output is synthesized and placed in the `mad_synth` structure. The structure is sent to either a file or the standard output. Finally, the input file is unmapped from memory, and the various structures are deallocated.

Frame decoding is performed in a number of steps. First, the header of the frame is read and the length of the current frame is determined. Then, frame data is parsed from the input stream. Data for successive frames does not synchronize with its respective header and overlap, as shown in Fig. 6. Thus, the data header may point to data preceding it. This data is copied into a buffer in the the `mad_stream` structure, called `main_data`, before decoding. Copying and the decoding of the data is performed in three stages: `load`, `decode`, and `preload`. In `load` stage, the data in the current frame is copied from the input stream into `main_data`. In `decode` stage, the data in `main_data` is decoded. In `preload` stage, the data associated with the next header is moved from one part of `main_data` to another, in preparation for processing of the next frame.

The first step in porting MAD to the Hyperprocessor's programming model is to cluster instructions into tasks. We have started an effort to develop efficient clustering techniques and will report our findings in a future article. For MAD, we defined tasks, simply, to be top-level functions. The next step is to generate a Sarek program to schedule the execution of tasks. This is a relatively simple step since the `main` function of MAD consists, for the most part, of calls to other

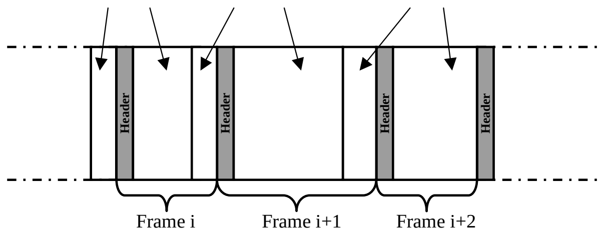


**Figure 6: Frame data in the input stream.**

```
Init (out mad_decoder);
Map (in mad_decoder, out mad_decoder);
while(end_of_buffer) {
   Header_Decode(in mad_decoder, out mad_decoder);
   Sideinfo(in mad_decoder, out mad_decoder);
   end_of_buffer = Frame_Decode(in mad_decoder,
                                out mad_decoder);
   Synthesis(in mad_decoder, out mad_decoder);
   Output(in mad_decoder, out mad_decoder);
}
DeMap(in mad_decoder);
Finish(in mad_decoder);
```

**Figure 7: The initial Sarek program for MAD.**

functions. The initial Sarek program, derived from the `main` function, is depicted in Fig. 7.

The task `Init` corresponds to the function that creates and initializes all data structures contained in `mad_decoder`. A pointer to this structure appears as a parameter to each functions in the MAD program, and thus, as a parameter to each task in the Sarek program. Since each task reads and writes variables to/from this structure, the pointer is designated as both input and output to each task.

While this initial Sarek program resembles the `main` function of the program, the use of a pointer to `mad_decoder` as input and output arguments to each task results in two problems.

The first problem is *false dependences* introduced by the use of a single pointer to reflect the dependences among the tasks - in this case a pointer to `mad_decoder`. False dependences arises when tasks, executing in parallel, use different parts of `mad_decoder`. The use of one pointer as both input and output causes all tasks to serialize, thus limiting possible parallelism among tasks.

The second and more serious problem is introduced by the use of pointers as parameters to tasks. In order for a task to write to a memory location in a buffer or a structure, the pointer to this memory location must be an `in` argument to the task, even if the task is not reading the memory location. This creates an unnecessary true dependence, which is caused by the pointer. This, in turn, makes hardware renaming inapplicable, since every task using the pointer must get the original copy of the pointer value as a result of this "true" dependence. Furthermore, even if the pointer is removed as an `in` parameter, through compiler transformations, the hardware will rename the value of the pointer in the corresponding register, not the data pointed to by it, which is in memory.

Consequently, we introduce a series of code transformations that are applied to both the Sarek program as well as tasks to overcome these problems. The transformations are based on well-known compiler optimizations and can be easily incorporated into the Sarek language and `C` compilers. These transformations are: *parameter de-aggregation*, *buffer privatization*, *buffer replication*, *task splitting*, *code hoisting*, and *loop unrolling*.

### 3.1.1 Task parameter de-aggregation

The purpose of task parameter de-aggregation is to expose the elements of structures in the parameter list of tasks, thus eliminating false dependences, and allowing the hardware to rename these parameters when appropriate. This is done by recursively expanding pointers to structures by components, until all task parameters are of primitive types (e.g., `int`, `float`, `int *`, etc.).

For each parameter, the direction of access (in and/or out)

```
Header_Decode (in mad_decoder, out mad_decoder);
```

(a) Before de-aggregation

```
Header_Decode( in this_frame,
               out next_frame,
               in ptr,
               ...);
```

(b) After de-aggregation

**Figure 8: An example of parameter de-aggregation.**

must be determined. We compute, using standard dataflow analysis techniques, upward exposed uses and downward exposed writes for each parameter in a task body. If there is an upward exposed read of a parameter, the parameter is designated as input; if there is a downward exposed write, the parameter is designated as output.

Fig. 8 shows the `Header_Decode` task header after transformation. It shows two (of many) elements of `mad_decoder` now appearing as task parameters. The first two parameters, `this_frame` and `next_frame`, are integers. Thus, the hardware is able to eliminate false dependences, from the use of this variable, among tasks. The third parameter, `ptr`, is a pointer to a buffer in memory, and as described above, causes a renaming problem. This problem must be resolved by renaming during the compilation step and is called *buffer privatization*. This transformation is described in the next subsection.

### 3.1.2 Buffer privatization

Buffer privatization is illustrated using the example in Fig. 9(a). Consider two tasks in the body of a while loop. The two tasks use the buffer `buff`. Output and anti dependences prevent instances of the tasks in one iteration from executing in parallel with instances in another iterations. These dependences result from the use of the same buffer area in memory by all iterations. Since the buffer is written to at the beginning of each iteration and is read from in the remainder of the iteration. It is possible to break the output and anti dependences by *privatizing* the buffer, i.e., by giving each iteration a private copy of the buffer, as shown in Fig. 9(b). A new buffer is allocated at the beginning of each iteration using the new task `Init`, and is deallocated at the of an iteration using the new task `Finish`. The hardware renames the parameter `buff` in each iteration of the loop, but now along with a corresponding private buffer[2]. The task `Finish` is guaranteed not to complete until all tasks in an iteration are complete through the addition of artificial dependences.

In order for privatization of a buffer `buff` to be legal, it is necessary that every read to a section of `buff` be dominated by a write to the same section of `buff` in the same iteration. This transformation is similar to array privatization [16], which is successful in the context of automatic parallelization of loops [6]. Thus, the transformation requires buffer section analysis [16] as well as interprocedural alias analysis, similar to that employed in [13].

This transformation is particularly effective in MAD because buffers within the `mad_stream`, `mad_frame` and `mad_synth` structures are re-used for the decoding of each frame.

### 3.1.3 Buffer replication

In some cases, not all reads to sections of a buffer are dominated by writes to the same sections in an iteration, making buffer privatization inapplicable. Nonetheless, parallelism can be introduced by overlapping the execution of tasks in one iteration with the execution of tasks in subsequent iterations. The transformation used to accomplish this is called *buffer replication*.

Buffer replication is illustrated using the example in Fig. 10(a). Consider the three tasks, `Load`, `Decode`, and `Preload` that all use the buffer `main_data`. They are part of the `Frame_Decode` component of MAD. `Load` writes only to the second half of `buff`; `Decode` reads the entire buffer; `Preload` reads the second part of `buff` and writes to its first half. Privatization is not possible because in an iteration, `Decode` reads buffer data written in the previous iteration by `Preload`. The dependences among the tasks cause their execution to serialize as shown in Fig. 11(a).

In order to achieve some parallelism, the `main_data` buffer is replicated and copied into the buffer `temp_data` in `copy`, as shown in Fig. 10(b). This copy is given to `Decode`, breaking the anti-dependence between `Decode` and `Preload`, and thus allowing them to execute in parallel. It should be noted that a new copy of `temp_data` is allocated every iteration of the loop, and that the hardware automatically renames `temp_data` every iteration. The execution of the tasks after buffer replication is shown in Fig. 11(b). There is now overlap among `Decode` task instances in different iterations because each instance uses a different buffer.

Buffer replication requires the same set of analysis required for buffer privatization, namely buffer section analysis and interprocedural alias analysis [13, 16].

### 3.1.4 Task splitting

This transformation aims to increase the parallelism among tasks by splitting a task into multiple tasks. For example,

[2]It is also possible to allocate a private copy of the buffer per processor, but this would require knowledge of the number of processors, which may or may not be the case.

```
while (...) {
    TaskA (out buff, ... );
    TaskB (in buff, ... );
}
```

(a) Before privatization

```
while (...) {
     Init (out buff);
     TaskA (out buff, ... );
     TaskB (in buff, ...);
     Finish (in buff, ... );
}
```

(b) After privatization

**Figure 9: An example of the buffer privatization transformation.**

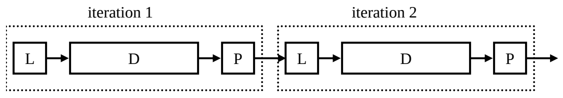
```
while (...) {
      load (out main_data, ... );
      decode (in main_data, ... );
      preload (in main_data, out main_data, ... );
}
```

(a) Before replication.

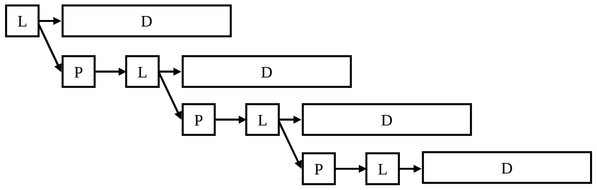
```
while (...) {
      load (out main_data, ... );
      copy (in main_data, out temp_data);
      decode (in temp_data, ... );
      preload (in main_data, out main_data, ... );
}
```

(b) After replication.

**Figure 10: An example of the buffer replication transformation.**



(a) Execution of tasks is serialized by dependences.



(b) Parallel execution is achieved by buffer replication.

**Figure 11: Task execution without and with buffer replication.**

the `Frame_Decode` task consists of calls to three functions; `Load`, `Decode`, and `Preload`, that perform the actions described in the previous section. However, these functions are not tasks. Thus, to extract the parallelism and to allow for other transformations to take place, the `Frame_Decode` task is split by transforming these three functions functions into separate tasks that are now part of the Sarek program. This transformation is similar to function inlining, except that it inlines functions into tasks.

#### *3.1.5 Code hoisting*

A task writes its output parameters by making calls to the `writeArg` function. Since a task cannot start until all its input parameters are available, it is desirable to write the output parameters of a task as early as possible to allow waiting tasks to proceed. Thus, we apply code hoisting [11] to the body of a task to move calls to `writeArg` to the earliest point possible.

#### *3.1.6 Loop unrolling*

The hyperprocessor architecture supports out-of-order speculative execution. Nonetheless, we apply loop unrolling to the main loop in the MAD Sarek program in order to increase the amount of task parallelism. The body of the main loop is copied as many times as the number of processors used ($P$), and the loop condition is checked every $P$ iterations.

### 3.2 FMR

FMR is an audio application that performs FM demodulation on a 16-bit input data stream, producing a 32-bit output data stream. The input stream consists of data packets of 1536 bytes each. The application primarily performs a sequence of operations, such as CIC low pass filtering, FM demodulation, and IIR/FIR de-emphasis on each input packet to produce the output.

The high-level steps of FMR is shown in Fig. 12. These steps are performed in the `main` function by 70 calls to 16 functions that are defined in the program. Thus, deriving a Sarek program for this application is straightforward. The Sarek program consists of a loop in which 70 task calls are made, corresponding to the 70 functions.

The main data structures used in the program are a set of buffers that are used to store and process each input packet. Pointers to these buffers are passed as arguments to the various tasks. The same set of transformations used for MAD

```
open input file
open output file
while(!eof) {
    get data from input file
    qamdemodulation
    filtering
    decimation
    fm demodulation
    deemphasis
    filtering
    decimation
    send to output file
}
close all files
```

**Figure 12: The overall operation of FMR.**

are also used for FMR.

In FMR, some parallelism can be achieved among the tasks in one loop iteration because the same tasks are applied to different intermediate buffers to produce different output buffers. However, some temporary buffers are re-used in a loop iteration, and buffer privatization within a loop iteration (as shown in section 3.1.2) is necessary.

## 4. EVALUATION AND ANALYSIS

We have developed a timed functional model of the Hyperprocessor in C++ with SystemC 2.0.1, in about 6,000 lines of code. The model reflects the overall structure of the Hyperprocessor, with a Control Processor, Task Dispatcher, Universal Register File and some PUs, with associated PU caches and shared or distributed memory.

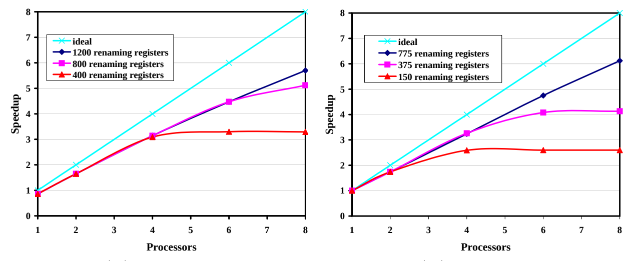
The model is parameterized to allow architecture exploration. The overall configuration and parameters are specified in a text file that is interpreted at run time to instantiate the desired configuration. This is a very useful feature for speed, since the model need not be recompiled for every configuration explored. Parameters to the model include: the number and type of PUs, the size of the URF, the number of renaming registers, the cache and memory configuration, and the latencies and cycle times for memory access, shared register access. Parameters also specify the relative speed of the CP, the TD, and the PUs.

The memory hierarchy is modeled at the transaction level. Each PU can be configured with a cache and a combination of local and global memory. The interconnect adds a constant delay, and the memory model implements a simple contention mechanism, where the requests are enqueued in order and dequeued at a given rate. The model of URF contention is similar. The caches are write-through, and thus memory always contains the most up-to-date copy of data. Cache lines need not invalidated on every write to maintain consistency; rather, caches can be maintained consistent by invalidating a cache at the end of task execution on the corresponding processor.

The model produces various statistics, including: the length of the simulation, the number of instructions for each ISS, number of read/write for the URF, the memories, number of cycles spent waiting for I/O, average latency for the memory operations, etc.

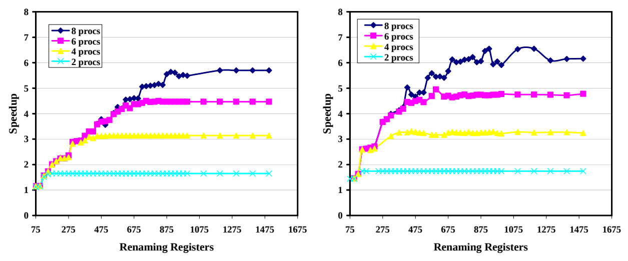
We also have developed a tool to compile Sarek to HASM. The tool does not perform any optimizations, but its functionality is sufficient so that we do not have to write assembly-level code manually. The tasks themselves are compiled for ARM using a Linux-to-ARM cross-compiler version of GNU's GCC. The model loads into memory the ELF object file.

We have used these tools to compile and simulate the exe-



(a) MAD (b) FMR

**Figure 13: The speedup of the MAD and FMR applications.**



(a) MAD (b) FMR

**Figure 14: Impact of the number of renaming registers on speedup.**

cution of the MAD and FMR applications. For these experiments, we use the ARMulator distributed with GNU's `gdb` [3] to implement an ARM Processing Unit. The ARM ISS has been slightly modified to allow multiple instantiations in our model, and to run under SystemC control. We also use a simple round-robin scheduler in the TD unit. We make use of 8-way set-associative caches with cache block size of 8 bytes and only global memory. We assume 4 ports to the URF (and thus renaming registers), and similarly 4 ports to the global memory.

The MAD application is used to decode an input MP3 song file, which consists of 126 frames, with 2 channels, 22050 Hz sample rate, and a 40 Kbps bit rate. It executes in 137312446 cycles on one processor. The FMR application is used to decode 21 input packets, each consisting of 1536 bytes. It executes in 112468135 cycles on one processor.

Fig. 13 shows the speedup of the two applications as a function of number of processors. The speedup of each application at $P$ processors is defined as the ratio of the execution time of the application on one processor to the execution time of the application at $P$ processors. The speedup is shown for various numbers of renaming registers (see below). The figure shows that each application exhibits scaling speedup, which is relatively close to ideal when the number of renaming registers is sufficiently large. The figure also shows that the speedup of MAD at 1 processor is slightly less than 1. This reflects the overhead that is incurred through the code transformations. For example, the transformed code of each application allocates and deallocates buffers during its execution, which does not occur in the sequential un-transformed application. Nonetheless, as the number of processors is increased, the benefits of parallelism outweigh this overhead.

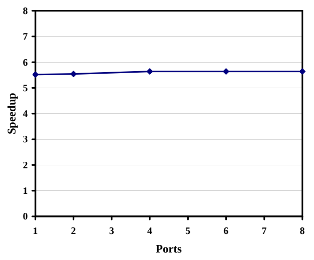
Fig. 13 also shows that speedup of each application depends on the number of renaming register; in general, the more renaming registers are available, the higher the speedup. Fig. 14 further explores this issue. It shows the speedup of MAD and FMR as function of the number of renaming registers for different numbers of processors. The figure shows that the speedup of each application experiences a noticeable increase up to a certain "breakpoint" point, then it remains relatively flat afterwards. This breakpoint is different for each application, and also for each number of processors. This behavior of application speedups can be explained as follows. The increase in the number of renaming registers allows more tasks to execute in parallel[3]. However, the increase in the number of registers will have an impact only if it enables the execution of more tasks. When all false dependences are removed by the addition of enough renaming registers, the speed of execution of each application is dictated by the true dependences among its tasks. The addition of more renaming registers does not lead to the execution of more tasks. Thus, the speedup improves up to the point where all false dependences are broken, and then remains flat. Similar to renaming registers, the availability of processors dictates the maximum number of tasks that can execute, and the breakpoint is also a function of the number of processors. Since the number of false dependences is different for each application, the breakpoint is also different for each application.

However, it can also be noticed in Fig. 14, especially for 8 processors, that the addition of more renaming registers sometimes decreases the speedup, rather than improves it. We attribute this to the impact the addition of registers has on the scheduling of tasks. Consider, for example, a task `A` that has 6 false inter-task dependences and a subsequent task `B` that has only 2 such false dependences. Assume that `B` is on the critical path of execution, i.e., the earlier it executes the sooner execution finishes. If there are only 4 renaming registers available, then `A` waits because of its 6 false dependences cannot be broken with 4 registers, but task `B` can proceed. On the other hand, if there are 6 renaming registers, then `A`'s false dependences are broken and it executes, causing `B` to wait. Since `B` is on the critical path, the execution time, and thus the speedup, is better with a smaller number of renaming registers. We have observed such scenarios in the execution of both applications, although they harder to describe due to their complexity. This suggests that the use of a simple round-robin scheduler in the TD unit is not the most prudent choice. We are exploring scheduling issues as part of future research.

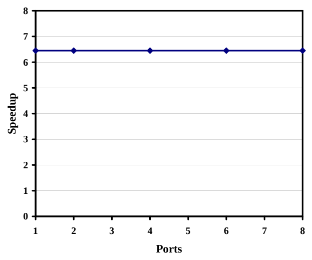
It is important to note that the potentially large number of renaming registers poses no performance bottleneck in the top-level of the Hyperprocessor. The granularity of tasks executing on the PUs is several orders of magnitude bigger than the URF register access time. Thus, URF registers are not likely to be accessed every cycle, and they need not be as fast as registers within the PUs. Indeed, as one indicator, we experimented with the impact of the number of access ports to the URF. In both applications, changing the number of access ports from 1 to 8 had negligible impact on performance. This is depicted in Fig. 15, which plots the speedup of each application at 8 processors as a function of the number of ports to the URF.

Finally, the impact of the number of memory ports on the speedup of the applications at 8 processors is depicted in

[3]The renaming registers allow the hardware to break false inter-task dependences, and thus to issue more task instructions in parallel. The hardware stops issuing instructions when it runs out of renaming registers.

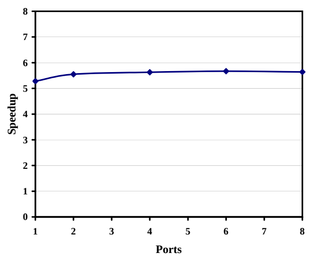


(a) MAD

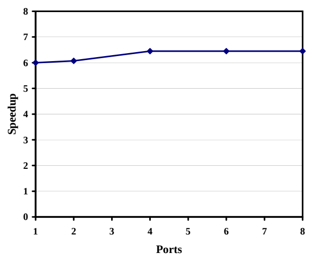


(b) FMR

**Figure 15: Impact of the number of URF ports on the speedup at 8 processors.**



(a) MAD



(b) FMR

**Figure 16: Impact of the number of memory ports on the speedup at 8 processors.**

Fig. 16. Similar to the URF, there is little impact of the number of memory ports on the speedup. This indicates that memory is not strongly contended for.

## 5. RELATED WORK

The Hyperprocessor uses superscalar technology, which was was pioneered by Tomasulo [15] in 1967. Hennessy and Patterson [8], and Smith and Sohi[14] offer excellent overviews of modern superscalar architectures.

The extension of superscalar principles to tasks instead of instructions and the associated scheduling was explored by Verians, Legat et. al [17, 19]. The core of their work is an efficient implementation of the instruction queue, which has to check on the operands availability to perform the out-of-order dispatching. However, they rely on a shared memory instead of a shared register file for the inter-task communication.

There exists a number of SoC systems that use multiple processing units for multimedia and other applications [1, 2, 4, 7]. However, they differ from the Hyperprocessor in their programming model.

Daytona's scalable DSP architecture [7] features a split-transaction bus for communication and cached semaphores for synchronization. The programming model [9] is also layered, separating tasks and top-level control flow. The dynamic scheduler is implemented in a run-time kernel and is configurable.

The picoChip [4] is a cascadable reconfigurable architecture of array processors intended for 3G wireless communications. However, in contrast to the Hyperprocessor, which is programmed in a quasi-sequential model, the picoChip is programmed using VHDL.

Cradle's Technologies's *3SOC* is a shared-address space multi-processor SoC. It consists of a number of processor clusters that are connected by two levels of buses [1]. The system provides 32 semaphore registers for synchronization, which must be explicitly used in a parallel program. In contrast, our Hyperprocessor architecture automates the synchronization of tasks through the URF, thus providing a programming model that is closer to sequential programming.

The code transformations used to improve the performance of programs for the Hyperprocessor build on a number of well-known compiler analyses and optimizations, including data flow analysis [11], array privatization [16], code hoisting [11] and loop unrolling [11].

## 6. CONCLUDING REMARKS

In this paper, we presented a novel architectural template for SoCs targeted for multimedia applications. Our architecture uses existing superscalar techniques to automatically exploit parallelism among tasks of coarse-grained computations. Our architecture allows tasks to be executed out-of-order; the hardware synchronizes tasks based on their true dependences and eliminates false dependences using register renaming. This hardware support gives rise to a programming model that is very close to sequential programming: individual tasks are expressed in a sequential language like `C`, while a task program written in a `C`-like language initiates the execution of these tasks.

We evaluated an instance of the template architecture using a functional and timing simulator of two realistic multimedia applications. The modularity of the applications makes it relatively straightforward to write an initial task-level program. We described a number of code transformations that are based on well-known compiler analyses. Our simulation results show that both applications exhibit good performance. Furthermore, the performance appears to be scalable, illustrating the architecture's potential.

Our future work will address a number of issues, including: the design and evluation of a memory hierarchy for the Hyperprocessor, task definition and formation, task scheduling, and system evaluation using industry-standard applications.

## 7. REFERENCES

[1] 3SOC documentation - 3SOC 2003 hardware architecture, cradle technologies, inc., march 2002.

[2] 3SOC programmer's guide, cradle technologies, inc., march 2002. http://www.cradle.com.

[3] `http://www.gnu.org`.

[4] `http://www.picochip.com`.

[5] `http://www.underbit.com/products/mad/`.

[6] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[7] C. Nicol et al. A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP. *IEEE Journal of solid-state circuits*, 35(2), March 2000.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[9] A. Kalavade, J. Othmer, B. Ackland, and K. J. Singh. Software environment for a multiprocessor DSP. In *Proc. of the Design Automation Conference*, 1999.

[10] F. Karim, A. Nguyen, S. Dey, and R. Rao. On-chip communication architecture for OC-768 network processors. In *Proc. of the 38th Design Automation Conference*, 2001.

[11] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] J. Paul and D. Thomas. A layered, codesign virtual machine approach to modeling computer systems. In *Proc. of Design Automation and Test in Europe*, March 2002.

[13] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proc. of PPoPP*, 1999.

[14] J. Smith and G. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83:1609–1624, 1995.

[15] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of R & D*, 11:25–33, 1967.

[16] P. Tu. *Automatic Array Privatization and Demand Driven Symbolic Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[17] X. Verians, J.-D. Legat, J.-J. Quisquater, and B. Macq. A new parallelism management scheme for multiprocessor systems. *Lecture Notes in Computer Science*, 1557:246–256, 1999.

[18] D. Wall. Limits of instruction-level parallelism. In *Proc. of ASPLOS*, pages 176–189, 1991.

[19] J.-J. Q. X. Verians, J-D. Legat. Extension du principe superscalaire au traitement de blocs d'instructions. In *ASTI 2001.*