PipeIt: A Pipeline Programming Framework for Embedded Processor Array Systems-on-Chip

Dimitris Syrivelis, and Spyros Lalis
Computer and Communications Engineering Department, University of Thessaly, Volos, Greece

Abstract— *This paper presents the* PipeIt *framework for developing pipelined applications targeted at tightly-coupled processor arrays on a chip. The framework includes a component programming and wiring model, a runtime environment, and a corresponding toolchain. It enables the programmer to develop applications in a high-level manner, structuring the code at the finest possible/meaningful level of granularity, without caring about how this will be deployed and executed. At runtime, the stages of the pipeline are distributed among the available processors. This arrangement can be changed dynamically when another* PipeIt *application needs to be executed concurrently. We discuss and demonstrate a complete embedded system prototype.*

Keywords: embedded processor arrays, application-level pipelining, dynamic load balancing, reconfiguration

1. Introduction

The promising performance and better physical as well as financial scalability, have motivated researchers to improve all aspects of multicore computing both for high-end and embedded systems. At the architecture level, approaches range from loosely-coupled processors interconnected via ethernet on different nodes, to tightly-coupled processors on a chip that are typically assigned with dedicated tasks and are connected together via high end dedicated links without any arbitration [1]. Platform differences in conjunction with the wide polymorphism in terms of applications also lead to a variety of partitioning and communication schemes. In turn, these can be supported by different tools. Of course, the type of computation at hand may naturally favor a certain scheme. Another important aspect is whether a multicore system is used in a dedicated fashion, or in the context of an open computing environment. In the former case, the optimal partitioning of the computation that will lead to the best possible performance can be decided at the design phase. On the contrary, in the latter case, new tasks may appear at any point in time and the available resources must be used opportunistically to boost performance.

In this paper we present the *PipeIt* framework which provides support for building and deploying application-specific pipelines on tightly-coupled distributed memory parallel processor arrays (PPAs) [1] for embedded systems in the context of an open, general-purpose computing environment. *PipeIt* includes a component and wiring model, a runtime backend that is appropriately customized for and integrated with the target execution environment, and a corresponding front-end compiler that generates the ultimate source code and builds scripts so that a regular toolchain can then be used to produce proper executables. A custom loader is used to deploy *PipeIt* application stages at runtime, and the initial arrangement can be changed at any point in time if the system workload changes. This is achieved *without* requiring PPA cores to feature heavyweight OS support.

The main contributions of this work are: i) the modular application design approach that enables the reuse of basic/-common pipeline structures; ii) seamless pipeline execution with support for the efficient dynamic reassignment of stages to available cores; iii) the ability to invoke a pipelined computation via a simple library call from within a conventional application; iv) a prototype implementation of all the development tools along with with an emulation environment for debugging and accessing expected performance of the pipelined computation; and v) an implementation of a well-known application on an FPGA-based PPA prototype.

2. *PipeIt* Target Platform

With *PipeIt* we wish to support the pipelining of typical CPU-intensive computations of embedded applications which operate on data block streams, e.g., cipher, (de)compression or encoding/decoding algorithms.

The target platform is tightly-coupled distributed memory parallel processor arrays (PPAs) [1], aimed for general-purpose embedded computing. PPAs typically feature reconfigurable, ultra fast and dedicated interconnections that introduce a small overhead for data block transfer. The individual processing elements are rather resource constrained, with memories of a few Mbytes, and cannot host proper OS support or heavyweight runtime environments. PPAs are currently used as dedicated coprocessors that may carry out one large computation at a time which is divided in an a priori known number of statically assigned tasks.

Our work assumes a special processor on the PPA (or an external processor connected to the PPA) which plays the role of the platform *master*, is interfaced to all platform peripherals, runs a proper OS/runtime, and is responsible for configuring the PPA to setup application pipelines as desired. In the general case, several pipelined applications may execute concurrently to each other but also other

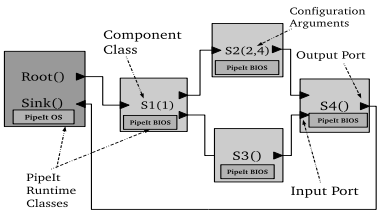

Figure 1: An indicative *PipeIt* pipeline

conventional applications running on the master processor. The system workload, in terms of both conventional and pipelined applications, may change dynamically.

3. The *PipeIt* Framework

To enable structured development of pipelined applications, *PipeIt* adopts a component model. Each component represents a pipeline stage that ideally should be executed using a separate processor. Components have a fixed number of input and output ports. They can be wired together, linking together output ports with input ports in a point-to-point fashion according to the desired data flow. The wiring of components is practically orthogonal to their implementation, and it is specified in a separate so-called configuration file. Basic checking is done so that interconnected ports handle data objects of the same size. During execution, each component blocks until data is available, processes data, and writes data to its output, in an endless loop.

The only component that has no input ports, i.e., does not wait for data to arrive from another component, is the pipeline entry point, called *root*. The *root* component typically reads data from an external source, such a file, special memory location or a special device. Similarly, the only component that has no output ports, i.e., does not send data to another component, is the pipeline exit point, called *sink*, which typically writes data to an external destination. The root and sink both run on the master processor (using two different threads), enabling the seamless integration of the pipelined computation with the rest of the system. Figure 1 shows an indicative *PipeIt* pipeline. Note that it is possible to have branches in order to introduce data parallelism inside the pipeline.

3.1 Component and communication model

A *PipeIt* component is coded as a C++ object, in a separate file with the same name. Each component class must be defined as a subclass of a *runtime type* class, which features two virtual functions, *config* and *exec*, that must be overloaded. The *config* function is called once, before the execution commences, and must be used to declare the ports of the component and initialize its internal state. Different configuration strings can be passed to each component, making it possible to implement flexible initialization schemes, and allowing for component classes to be reused in the same or different *PipeIt* applications. The *exec* function must contain the component's data transfer and stage processing code. It is invoked from within the *PipeIt* runtime, in an endless loop.

Data transfer is performed using the *input* and *output* functions. These are inherited from the base runtime type class, and are mapped to the communication primitives of the respective runtime environment. Ports are addressed using a simple numbering scheme which is mapped by the *PipeIt* framework to appropriate target-specific addresses. In a nutshell, components receive and send data using the abstract *PipeIt* primitives and port ids without caring about the underlying platform details.

Memory allocation of data buffers must be done using the *pipeIt_malloc* function provided by the runtime. This is necessary because *PipeIt* needs to control data access and transfers in order to perform component migration safely during pipeline restructuring. For the same reason, static declarations of data transfer blocks are not allowed.

As an example, we give the code of a simple *PipeIt* component that receives an integer from its input port, increments it, and forwards it to its output port:

```
class IncInt : public PipeItOs
{
  public:
    IncInt() {;}

    void config(int argc, char *argv[]) {
      data = pipeit_malloc(sizeof(int));
      pipeit_add_input(0, sizeof(int));
      pipeit_add_output(0, sizeof(int));
    }

    void exec(void *d, int size) {
      pipeit_input(0, &data, sizeof(int));
      *data = *data + 1;
      pipeit_output(0, &data, sizeof(int));
    }
};
```

Data is passed from one component to another by writing and reading proper output and input ports. However, some variables may need to be accessed *only* between the root and sink components, hence need not travel through the entire pipeline. A separate mechanism, called the *context queue*, is employed to keep track of these values in synch with the pipeline. Before writing data into its output, the root adds a context entry with the proper values, and, conversely, the sink removes the next context entry before attempting to read data from its input. In our implementation, the root and sink threads use a shared memory FIFO queue.

3.2 Runtime classes

PipeIt introduces runtime support mainly for two reasons: i) to confine the programmer during the development of a component to the execution environment and available resources of the target processing element; and ii) to be able to dynamically reconfigure the component placement in a seamless fashion.

There are two radically different execution environments, the master processor which runs a proper operating system,

and ordinary PPA processors running a small basic I/O system (BIOS) that is a custom implementation. In both cases *PipeIt* adds a thin layer providing a set of generic data transfer primitives, optimized for the respective environment. There are three different *runtime type* classes, *PipeItOS*, *PipeItBIOS* and *PipeItOSLib*, which can be used to develop components. Each reflects a different flavor of the *PipeIt* runtime support, as follows.

The *PipeItOS* class is used for components that will execute on the master processor, having access to the full functionality of a proper OS. This runtime class is used for the root and sink components, which may contain system calls and access peripherals. The generated code is an autonomous executable that runs on the master processor under the full-fledged OS and uses a separate *POSIX* thread for running each component.

The *PipeItBIOS* class is aimed at components that should run on ordinary PPA cores on top of the *PipeIt* BIOS. In this context the programmer may only perform CPU intensive computations, read data from input ports and write data to output ports. Attempts to use a non-existing runtime feature will cause the compilation of the component to fail. The default for such components is for them to execute on a dedicated PPA core. However, as a result of pipeline reconfiguration, several such components can be placed on the same PPA core or even on the master processor.

The *PipeItOSLib* class has a similar functionality to *PipeItOS*, but it does not result in the generation of an autonomous executable. Instead, it produces code that enables the pipelined computation to be invoked from within an external application context, much like a library. In this case, the root and sink components execute as *POSIX* threads and must establish an appropriate communication channel with the application, based on the arguments of the *config* and/or *exec* functions. Any IPC mechanism can be used for this purpose. The corresponding initialization code is placed in a routine named according to a certain convention, and this routine must be invoked from the application before initiating communication with these components.

3.3 Configuration language

The wiring of each *PipeIt* computation is specified in a separate configuration file. Configurations are expressed using three elements: component declarations, port connections and composites.

Components are declared using the class names of the respective implementations, optionally giving a configuration string that can drive initialization. The configuration string is not interpreted by the *PipeIt* framework; it is passed "as is" to the component, via a call to its *config* function. Input and output ports are denoted in brackets placed at the left and right hand side of a component name, respectively. Each connection is denoted by a right arrow, starting from an input port and pointing to an output port.

To enhance the structure of complex computations, and to enable the reuse of common sub-structures, configurations can be grouped into so-called composites which export their input and output ports. Composites have a so-called execution type, for which there are two options. The *PipeItMaster* type is used for composites that will run on the master processor; their components must extend the *PipeItOS* or *PipeItOSLib* class. The *PipeItArray* type is used for composites that should run on ordinary PPA cores, and all of their components must extend the *PipeItBIOS* class. A *PipeIt* configuration file has exactly one *PipeItMaster* composite and an arbitrary number of *PipeItArray* composites.

As an example, the application shown below uses two composites to increment an integer value twice. The *IncIntTwice* composite of type *PipeItArray* uses two appropriately connected instances of the *IncInt* class. One instance is declared explicitly while the other is declared implicitly, via the class name. The *MyApp* composite of type *PipeItMaster* contains a *MyRoot* and a *MySink* component (the code of those component is not shown here).

```
PipeItArray IncIntTwice {
  inc :: IncInt();
  input[0] -> [0]inc;
  inc[0] -> [0]IncInt()[0] -> output[0];
};

PipeItMaster MyApp {
   r :: MyRoot("42");
   s :: MySink();
   r[0] -> output[0];
   input[0] -> [0]s;
};

MyApp[0] -> [0]IncIntTwice;
IncIntTwice[0] -> [0]MyApp;
```

Local (intra composite) connections are declared within the respective scope, while global (inter composite) connections are defined at the end of the configuration file. The keywords *input* and *output* refer to the input and output ports of the composite. Also, in this example, the computation takes its input via the configuration string "42" passed to the *MyRoot* component, hence the *MySink* component will receive the value 44.

3.4 Dynamic load balancing support

The pipeline structure of a *PipeIt* application is designed assuming that all the components (stages) of the pipeline will be executed on a dedicated PPA core. However, at deployment time, there may not be as many processors available, either because the system does not have them in the first place or because some processors are already being used for other applications. In addition, during the execution of the application, new tasks may arrive and exiting tasks may finish. Thus a dynamic restructuring of the application pipeline is needed in order to release some processors, or, conversely, to exploit some of the processors being released.

To enable the flexible and concurrent deployment and execution of pipelined applications *PipeIt* comes with built-

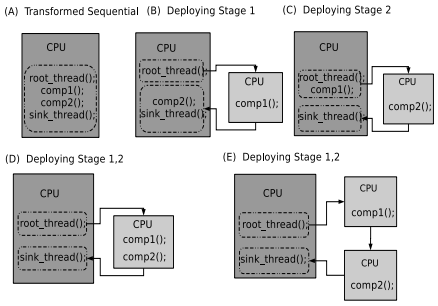

Figure 2: Configurations for a pipeline with 4 components

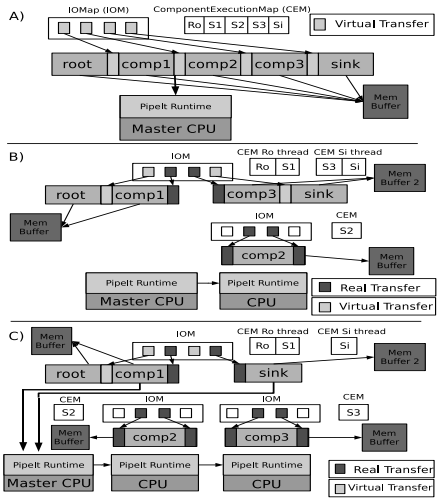

Figure 3: Structures for supporting pipeline restructuring

in support for dynamic load balancing. Specifically, the runtime can assign the component of a pipeline on the same or different processors in a transparent way. The assignment obeys the following rule: if two components are assigned on the same processor, every component between them must also be assigned on that processor. Figure 2 shows all such configurations for a pipeline with four components, including the root and sink.

The components to be executed on each processor are specified using a so-called *ComponentExecutionMap*, which is disseminated from the master to the PPA processors via a simple protocol. On each processor, *PipeIt* uses a simple scheduler to execute all co-located components sequentially. No real data transfer is performed between co-located components. Instead, both ends of each local link share a data buffer which is accessed from within the respective I/O calls. The I/O behavior of each component is controlled via a so-called *IOMap*, which indicates whether the *input* and *output* calls should perform a remote or local/virtual data transfer.

The initial configuration of the pipeline is established as follows. When the computation is first deployed, *PipeIt* runs the pipeline for a number of iterations sequentially on the master processor. During this initial execution phase, an appropriately instrumented version of the *exec* call is employed, through which *PipeIt* collects information that can be used to estimate the processing overhead of each component. Next, the component assignment scheme that will be used given the available number of processors is decided. Finally, the required processors are allocated, and the application code (containing the code for all components) along with the corresponding *ComponentExecutionMap* and *IOMaps* are loaded on each processor.

The pipeline can be reconfigured at any point in time during execution. This makes it possible to adapt to changing workload conditions, exploiting processors that become available or releasing processors for the benefit of other applications. Load balancing is guided using a system service which must be inquired periodically to determine the most appropriate configuration. In our current implementation, where frequent monitoring introduces considerable overhead, the rate at which this need to be done is specified by the programmer during compilation. If a new processor and/or component assignment is determined to be more beneficial, the *PipeIt* runtime performs the respective processor allocation and loading, updates the *ComponentExecutionMap* and *IOMaps*, pushes this information down the pipeline, and proceeds with the execution.

Figure 3 shows three indicative configurations for an application with five components, together with the corresponding data transfer and execution mappings. In case (A) all components execute sequentially on the master processor, and communication is done using shared buffers. In the case (B) the third component is set for execution on a PPA processor, and the output mapping of the second component as well as the input mapping of the fourth component are set to invoke the appropriate communication primitives to send/receive data between the master processor and the PPA processor. Finally, case (C) depicts the deployment of second and third component on distinct PPA processors.

3.5 Application development and tools

The developer must first provide at least the skeleton for each component, and then write the application configuration file. The *PipeIt* compiler parses the file, creates the appropriate data structures used to configure all aspects of the pipeline structure, and generates corresponding flavours of *pipeit.h* files, to be included by convention in each component implementation. These header files contain static declarations of various required variables, including the transfer bitmaps and profiling structures, as well as support structures to map the port numbers onto the platform-specific addressing primitives for each component. At this point, regular development toolchains can be used to compile the generated code for the target and the emulation platform.

The default mode is to produce code for all components to execute on the master processor environment, and for

all components except the root and sink to execute on an ordinary PPA processor environment. The compiler also accepts a hint in terms of preferred component co-location for the case where there are not enough PPA processors or the local memory of a PPA processor cannot host all components; in the latter case, different executables will be generated for different sets of PPA processors.

The *PipeIt* toolchain can also be used to generate executables for emulated execution on a Linux host. In essence, the master and PPA processors are emulated using distinct processes and interconnections are emulated via unix named pipes. The number of available PPA processors is specified by the user. Running an application in emulation mode simplifies debugging. Moreover it provides a computation to communication ratio estimate and enables the use of sophisticated profiling tools like *gprof* to guide component partitioning and co-location preferences. Another motivation for using the emulation mode is to assess the expected performance on the target platform for various pipeline configurations. Of course, if the emulation host has a radically different architecture from the target platform, it might not be possible to make an accurate estimation.

4. *PipeIt* Prototype and Applications

Our prototype is a custom PPA system implemented on a FPGA as a system-on-chip. The hardware platform is an Atmark Techno Suzaku [2], which features a Xilinx Spartan 3 FPGA along with off-chip peripherals. For the master and ordinary PPA cores we use the *Xilinx Microblaze* soft processor, a classic 32-bit RISC architecture. Microblaze features a fast bus architecture named Fast Simplex Links (FSL). This is a dedicated 32-bit wide unidirectional point-to-point communication channel, which does not need arbitration, provides hardware support to distinguish between data and control communication, and supports blocking/non-blocking asynchronous access.

The master processor is interfaced to all platform peripherals and is responsible for running a customized version of the *uClinux* embedded operating system [3], achieving 25.29 *BogoMIPS*. The PPA processors have only local memories and are connected to each other with FSL links. The entire system can be dynamically reconfigured at runtime using special support which we have developed in previous work [4]. For the purpose of this work, we have also developed an OS service that exports information about the number of available PPA cores and master cpu usage via *proc* filesystem, used by the *PipeIt* runtime in order to determine the component placement for application pipelines.

4.1 Applications

As a proof of concept application, we have implemented a *PipeIt* versions of the Secure Hash Algorithm and HMAC authentication code. We also integrated this implementation in the *Crypto* library, using the *PipeIt* library mode.

The SHA1 code employs 4 different functions which perform the same amount of computation on a data block, doing 80 sequential invocations with varying parameters in total (the original code is highly optimized, using inline functions etc). Hence, the *PipeIt* implementation is based on 6 component types (for the root, the sink, and each function) which are used to construct a pipeline of 80 components plus the root and sink. Below we list a simplified version of the source code of a typical component type, followed by an excerpt of the configuration file:

```
class R0: public PipeItBIOS
{
   public:
     R0(){;} ~R0(){;}
   struct Data *d;
   int arg1, arg2, arg3, arg4, arg5, offset;

   void config(char * args) {
    d=pipeit_malloc(sizeof(struct Data));
    pipeit_add_input(0, sizeof(Data));
    pipeit_add_output(0, sizeof(Data));
    parse_args(args);
  }
  void exec(void *d, int size) {
    pipeit_input(0, (void *)d,sizeof(Data));
    R0Calc(d+arg1,d+arg2,d+arg3,d+arg4,d+arg5,offset);
    pipeit_output(0, (void *)d, sizeof(Data));
  }
  void R0Calc(void *p1, void *p2, void *p3,
              void *p4, void* p5, int offset) {
    ...
  }
};


 PipeItArray SHA1 {
   input[0] -> [0]R0("a b c d e 0")[0]
   ...
  -> [0]R4("b c d e a 79")[0] -> output[0]
 };

 PipeItMaster SHA1App {
  SHA1Root() -> output[0];
  input[0] -> [0]SHA1Sink()
 };

 SHA1App[0] -> [0]SHA1;
 SHA1[0] -> [0]SHA1App;
```

The HMAC computation is based on SHA1. Indeed, the *PipeIt* version of HMAC *reuses* the *PipeIt* version of SHA1, as it can be seen from the corresponding configuration file:

```
 PipeItArray SHA1 {
   input[0] .. -> output[0]
 };

 PipeItMaster HMACApp {
   HMACLibRoot() -> output[0];
   input[0] -> [0]HMACLibSink();
 };

 HMACApp[0] -> [0]SHA1;
 SHA1[0] -> [0]HMACApp;
```

4.2 Experimental results

We tested the *PipeIt* version of SHA1 on our platform as an autonomous application as well as as a part of an HMAC application (the HMAC keys are set once, at the beginning of the program). Both applications feed the pipeline with predefined data blocks in an endless loop, mimicking a

continuous stream. Taking advantage of the library-oriented execution mode of *PipeIt*, we also integrated the pipelined version of HMAC (and SHA1) in the *Crypto* library.

In a first series of experiments, we performed measurements for the case where the pipelined computation runs: (i) only on the master processor; (ii) on 4 processors including the master; and (iii) on 5 processors including the master. In all cases, the system was unloaded, i.e., there were no other applications running at the same time. The results are shown in Table 1, including the performance of the original (highly optimized) sequential programs as a reference. The performance of the *PipeIt* HMAC is naturally dominated by the performance of SHA1.

The first observation is that the sequential execution of the pipelined versions introduces a notable overhead, performing at about $0.8x$ compared to the original code. This is because *PipeIt* explicitly invokes each component in a loop, thus it is not possible to optimize code, e.g., by using inline functions. The second observation is that the speedup achieved when using 5 processors is $4.45x$ compared to the sequential *PipeIt* execution, and $3.5x$ compared to the original version. The latter is far away from what is theoretically possible ($5x$), mainly due to the execution and communication overhead imposed by the *PipeIt* runtime for co-located components. Specifically, for SHA1, 12 components plus the root and sink are assigned to the master, while other processors are assigned 17 components each. The overhead increases as less processors are used and the number of co-located components increases, as demonstrated for the case of using just 4 processors, each having 20 assigned components (the master also runs the root and sink).

It is of course possible to boost performance by adjusting the number of application-level components (pipeline stages) to better fit the actual platform capabilities; in this case, by introducing fewer and more heavyweight components. To demonstrate this we created a second pipelined version of SHA1 with just 5 components in addition to the root and sink. Each component contains an optimized integrated version of the code of the components that were co-located in the 5-processor execution scenario. This program achieved a throughput of $253.9kb/s$ which roughly equals to a $4.6x$ performance improvement over the original sequential code, and $4.76x$ compared to the corresponding sequential *PipeIt* execution. The downside is that the 5-component pipeline is very coarse-grained and cannot possibly give a speedup greater than $5x$, *even if* the underlying platform features more cores. On the contrary, the 80-component pipeline version may theoretically reach a $80x$ speedup if run on a PPA with 80 (idle) cores. Another issue is that having fewer pipeline components/stages also limits the options of the runtime in terms of evenly distributing the amount of processing on top of fewer processors.

To verify the ability of our system to perform dynamic load balancing, we run two instances of the *PipeIt* SHA1 computation concurrently to each other, using a total of 5 processors including the master. The first instance is started on an idle system, exploiting all 5 processors as discussed above. The second instance is started at a later point, when the pipeline of the first instance is already running. As a result of dynamic balancing, each computation is assigned 2 processors in addition to the master, and the two pipelines are configured appropriately, having just the root and sink on the master and 40 components on each other processor. The throughput achieved by each computation is $76Kb/s$, giving a total of $152Kb/s$. For comparison, the performance achieved for a single computation with the same pipeline/-component configuration on an idle system is $78Kb/s$. This overhead is due to the fact that the master processor is shared between the (root and sink of) the two computations, and this contention leads to occasional pipeline stalls.

Computation	original	*PipeIt* seq	4 CPUs	5 CPUs
SHA1	$55.2Kb/s$	$43.1Kb/s$	$156Kb/s$	$193Kb/s$
HMAC	$54.1Kb/s$	$42.8Kb/s$	$154.9Kb/s$	$191Kb/s$

Table 1: Performance of SHA1 and HMAC

Having the pipelined version of HMAC (and SHA1) integrated in the *Crypto* library, makes it trivial to exploit it from within *existing* and *conventional* applications in a transparent fashion. As a proof of concept, we used *scp* (version 2 of the SCP protocol) to copy a $10MByte$ file over Ethernet from a PC connected on the same switch as the Suzaku board. Using the original HMAC/SHA1, the transfer was performed at $18.1Kb/s$ vs $22.4kb/s$ when using the *PipeIt* versions, giving an improvement of $1.23x$. The speedup is relatively small because HMAC accounts for a rather small part of the processing done by *scp* (which was left untouched). It is also important to note that since the conventional part of *scp* runs on the master processor, the *PipeIt* runtime deploys the HMAC/SHA1 pipeline on just 4 processors, using the master only for the root and sink.

To confirm that load balancing works better when the pipeline has a finer granularity, we repeated the same file transfer experiment using the 5-component pipeline version of HMAC/SHA1 discussed previously. In this case, the throughput was only $20.4kb/s$. Given that the *PipeIt* runtime decides avoid using the master (except for the root and sink), 2 out of the 5 (coarse-grained) components end up being executed on the same processor, resulting in an unbalanced distribution of the pipelined computation which in turn leads to a deteriorated performance.

5. Related Work

The Ambric PPA architecture [1] integrated with a master CPU that can run a full-fledged OS would be an ideal high-performance target for our framework. Currently Ambric uses a structured object programming model. The programmer separates the application into high-level processing objects which can be developed independently and can execute asynchronously with each other, at their own clock

speed, on their own dedicated processor core. We believe that the integration of PPA functionality in an OS context with dynamic load balancing is also important. To that end, our approach could be used to efficiently accommodate concurrently running applications on such platforms.

The component-based design of *PipeIt* has been heavily influenced by the Click framework [5], although this targets a different application domain, namely the modular implementation of router functionality. Click features *C++* objects and employs a configuration language for specifying a network of objects. The *nesC* language [6], introduced to support application development for resource constrained wireless nodes (motes), also relies on the notion of components and wiring configurations.

StreamIt [7] also introduces a high-level programming model, but for a broader application domain than *PipeIt*, including all types of applications that use a stream as an abstraction. In this case, there is no notion of a component configuration, instead components are explicitly interfaced to each other as a part of their implementation. The StreamIt compiler can automate tasks such as partitioning, static load balancing, layout, and memory management. However, to our knowledge, there is no support for balancing a computation at runtime.

Coarse-grained pipelining is addressed in [8], where annotations are proposed in order to perform the required code restructuring at the source level. Pipeline parallelism is also exploited in [9] using the techniques of Decoupled Software Pipelining [10], also employing thread-level speculation to opportunistically execute multiple loop iterations in parallel. While these approaches could be used for a PPA target, they both assume a homogeneous shared memory system where processors are a priori assigned to applications.

In the spirit of *PipeIt*, work in [11] also targets a dynamic computing environment where platform resources are not statically dedicated to computations. To achieve balanced execution of parallel applications, a user-level scheduler is proposed, which dynamically distributes tasks over a fixed collection of processes, which in turn are scheduled on a fixed collection of processors by the operating system kernel. Contrary to *PipeIt*, this approach requires a full-fledged OS on each processor in order to run OS-level processes and IPC mechanisms.

Finally, dynamic task partitioning methods have been proposed to deal with unstructured mesh problems [12], [13]. In [12] the tasks of a computation are developed using an appropriate programming model and a software framework, which features an extension called Mobile Object Layer(MOL) [14] that enables transparent task migration between processors at runtime. The authors have extended the MOL concept by adding load balancing routines that communicate with respective runtime support. In this case, the target platform is not a tightly coupled SoC and load balancing remains an application-level task, i.e., the programmer has to write code explicitly for this purpose.

6. Conclusion

We have presented *PipeIt*, a framework that supports the development of pipelined applications for embedded PPA targets in the context of an open, general-purpose runtime environment. Our vision is to have a system where cpu-intensive tasks of various applications are implemented as fine-grained pipelines (if appropriate) which are deployed on the available PPA processors in a flexible way, as a function of the current system workload. To that end, providing support for dynamic load balancing, without forcing the programmer to think about the platform constraints and without requiring full-fledged OS support on each PPA processor, is of major importance.

7. Acknowledgments

This paper is part of the 03ED918 research project, implemented within the framework of the “Reinforcement Programme of Human Research Manpower” (PENED) and co-financed by National and Community Funds (75% from E.U.-European Social Fund and 25% from the Greek Ministry of Development-General Secretariat of Research and Technology).

References

[1] M. B. et al., “A structural object programming model, architecture, chip and tools for reconfigurable computing,” in *FCCM*, 2007.

[2] *Suzaku Series*, Atmark Techno Inc, http://www.atmark-techno.com/en/products/suzaku.

[3] J. Williams, *The Microblaze-uClinux kernel port Project*, http://www.itee.uq.edu.au/ jwilliams/mblaze-uclinux/.

[4] D. Syrivelis and S. Lalis, “System- and application-level support for runtime hardware reconfiguration on soc platforms.” in *USENIX ATC*, 2006.

[5] E. K. et al., “The click modular router,” *ACM Trans. Comput. Syst.*, vol. 18, no. 3, 2000.

[6] D. G. et al., “The nesc language: A holistic approach to networked embedded systems,” in *PLDI*, 2003.

[7] W. T. et al., “Streamit: A language for streaming applications,” in *Computational Complexity*, 2002.

[8] W. Thies, “A practical approach to exploiting coarse-grained pipeline parallelism in c programs,” *MICRO*, 2007.

[9] M. J. B. et al., “Revisiting the sequential programming model for multi-core,” in *MICRO*, 2007.

[10] N. V. et al., “Speculative decoupled software pipelining,” in *PACT*, 2007.

[11] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” in *SPAA ’98*. ACM, 1998, pp. 119–129.

[12] K. B. et al., “A load balancing framework for adaptive and asynchronous applications,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 2, 2004.

[13] C. W. et al., “Parallel dynamic graph partitioning for adaptive unstructured meshes,” *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, 1997.

[14] N. C. et al., “Mobile object layer: a runtime substrate for parallel adaptive and irregular computations,” *Adv. Eng. Softw.*, vol. 31, no. 8-9, 2000.