OS Paradigms Adaptation to Fit New Architectures *

Xavier Joglar
Universitat Politècnica de Catalunya
Jordi Girona 1-3
Barcelona, Spain
xjoglar@ac.upc.edu

Judit Planas
Universitat Politècnica de Catalunya
Jordi Girona 1-3
Barcelona, Spain
juditp@ac.upc.edu

Marisa Gil
Universitat Politècnica de Catalunya
Jordi Girona 1-3
Barcelona, Spain
marisa@ac.upc.edu

ABSTRACT

Future architectures and computer systems will be heterogeneous multi-core models, which will improve their performance, resource utilization and energy consumption. Differences between cores mean different binary formats and specific concerns when dealing with applications. The OS also needs to manage the appropriate information to schedule resources to achieve the optimal performance.

In this paper we present a first approach in Linux to allow the application to give information to the OS in order to perform the best resource scheduling for the code characteristics (where it has to run). Based on the continuation model of the Mach microkernel and the device drivers of Unix-Linux system, the kernel can continue the execution flow from one core to another (i.e. from PPE to SPE in the Cell BE case). In this way, the OS can anticipate costly actions (for example, loading code or data) or reserve resources depending on task needs.

To reach our target, we adapt the operating system as well as modify the application binary to divide its code parts depending on their characteristics and where they have to run.

Experimental work has been done for x86 with MMX extension ISA as well as for PPC and Cell BE.

Keywords

Heterogeneous multi-core, OS loader, fat binary.

1. INTRODUCTION

Current technology trends move to new processors with specialized cores, like Cell BE [6] or the Intel Exoskeleton [11] with specific processor elements such as those which are GPU-based [7], or other accelerator elements, including DSP and FPGA units. In this scenario, programs must be redesigned in order to fully exploit this heterogeneity and improve their performance.

In this way, they can be built from components that work better in specific engines in a more energy-saving way. These specific cores are also suitable to execute specific parts of a program to improve its overall throughput.

From this preliminary point, some aspects must be analyzed and modified. In a first stage, programming and compiling parallel applications should be heterogeneity-aware, deciding and building the appropriate executing stream. Then, a specific management task must be done by the linker and the loader to compose and load the final executable binary, interpreting its new information and performing the appropriate actions. Finally, the operating system must manage, at runtime, the physical resources.

To take advantage of these specialized units, the OS must be able to schedule the code parts onto the most appropriate unit depending on the code requirements as well as the available execution units [5]. In this work, programmers can help the OS to manage the system resources more efficiently by giving some hints about the application behaviour. For example, if the programmer knows that a function or a loop has special characteristics, he can mark the code properly. That is why it is necessary to add some additional information in the executables and make the operating system able to manage and understand the given information.

Some requirements to achieve this OS improvement are:

- Having the application code portions in the binary (or bit stream map) corresponding to the platform where it has to run.
- Having a “fat binary” executable file containing the different binary parts of code [11].
- Allowing threads to run sequentially cross-ISA (i.e. from one unit to one of a different type) [9].
- Providing the OS with the requisite information to schedule a thread in the appropriate engine (depending on the ISA, processor affinity or system load).

* This work is supported by the European Commission in the context of the SARC Integrated Project (EU contract 27648-FP6), the HiPEAC European Network of Excellence and the Ministry of Science and Technology of Spain and the European Union (FEDER) under contract TIN2007-60625.

The main contributions included in this paper are:

- An extensible runtime fully compatible with both current and new architectures.
- A new loader to manage new binary format.
- New OS objects to hold application heterogeneity information.

The rest of the document is organised in the following way: in the next section, we describe the environment and introduce a brief idea of our target. In the third section we explain the modifications at application level as well as the API implemented to make the interaction between the user and operating system level easier. In section four we present the OS adaptations to be able to manage, store and use the new information. In section five we discuss our results and in section six, about the conclusions and future work. Finally, in section seven we give the references where readers can find more information related to our work.

2. OUR APPROACH

As mentioned above, hardware performance can be limited by its software resource management. So as hardware evolves, software has to evolve too, and vice versa. We thought that adapting the executable format to distribute application execution into different processor units would be an exciting chance to improve system performance.

Our proposal is to include additional information into the binary file so that the operating system can understand it and manage system resources as efficiently as possible [8].

We looked for a simple and portable format through different architectures. We considered that the best option would be to modify the ELF format [10] to make the transformation of the conventional ELF files easier.

We call our new extension Heterogeneous ELF (HELF) and we use the ELF sections to store the information that the operating system needs. HELF extension philosophy is based on the idea of embedding different binaries compiled for different ISAs in a single binary file.

The CESOF[1] extension used for Cell BE programs also allows different-ISA binary in a single ELF-format file. CESOF embeds compiled code for both the PPE and SPE in a single file. It uses several ELF sections to include the embedded SPE executable image with additional PPE symbol information.

The two main differences between HELF and CESOF are:

- The type of the new sections created for each piece of code compiled for a different ISA: CESOF creates read-only data sections and we create text sections[2].
- CESOF is focused on Cell BE; we want HELF to be adaptable to as many heterogeneous platforms as possible.

Programming models using the ELF format for heterogeneous multi-core are:

- The Cell Superscalar (CellSs) [1]: provides a simple, flexible programming model that can take advantage of the performance benefits from the Cell. CellSs is built from two principal components: the CellSs compiler and a runtime library. The compiler is a source-to-source translator that takes in an annotated C source file and produces a pair of source files, one for each Cell processor type (PPU and SPU). The runtime constructs a data dependency graph to schedule independent annotated functions to execute on different SPUs in parallel. CellSs uses CESOF.
- The EXO-CHI for the Intel Exoskeleton [11]: uses accelerator elements in a heterogeneous multi-threaded execution model and provides the tools needed to exploit the system performance and reduce power consumption.

The loader must be able to identify HELF files and classify the different parts of code depending on the information added by the programmer or, automatically, by the compiler to the ELF sections. In this way, the operating system will schedule them in the appropriate processor.

To achieve this, we have to consider three aspects:

1. How to adapt binary code in order to split it into different chunks of execution code[3].
2. How to modify the Linux loader in order to understand and load heterogeneous binaries.
3. Which new data structures have to be added to the kernel to store the new information needed.

For the moment, we are not focusing on data sharing, so some changes might be necessary to adapt our proposal to non shared-memory architectures.

When a code has to continue its execution in a specific core, for reasons such as being compiled for a special ISA or to obtain better performance in an accelerator, the only information that has to be provided is where it continues (an

[1] More information related to CESOF (such as diagrams or code examples) can be found in [2].

[2] For now, we are just considering self-contained functions, so that we do not concern about data.

[3] This is a programming and compiler concern, so we will not explain the details of this adaptation in this document.

address) and which input data it needs (the parameters). This is the same as what the explicit continuation model does [3].

We have implemented new library calls to specify the entry point and the unit identifier. Based on this information, the system is able to get the required data and prepare the loading and running environment in the new unit.

In the next sections we explain the application and the operating system adaptation to allow that.

3. THE APPLICATION SIDE

In this section we present our proposal at the application level. In particular, we explain the way to modify the source code of an application in order to reflect its heterogeneity, and the way its characteristics are reflected in the binary. We also show an example application to compare the original and the adapted version.

From the application point of view, we need to adapt binary files to mark the different code personalities.

The point of HELF executables is that the code has been classified according to its characteristics. The method we use to differentiate between these divisions is to create one section for each different piece of code, so that code and data inside a section are homogeneous and with a specific engine profile[4].

In this way, the operating system will detect these new sections and take the appropriate actions to exploit the heterogeneity of the hardware resources and so improve the application performance.

The execution flow of a HELF file is the same as that of the ELF, until the magic number is checked. At this point, because we have provided a new magic number for our files, if the file is a HELF executable, our loader is called instead of the ELF version.

In the next subsections we explain the different steps that an application follows from its source code until it is executed. Also, we introduce the user library we have implemented in order to take advantage of the new OS capabilities we have added.

3.1 Source code

It could be very useful for programmers to use some kind of library or compiler directive in order to decide exactly which chunks of source code they want to divide and where they want to execute each section (for example, specifying the kind of engine if it is possible). One possibility is the use of *#pragma* [1] and [4].

[4] Although it is also possible to have FPGA byte stream sections.

We will use as an example a matrix multiplication implemented for the Cell BE. The structure of the original code is shown in Figure 1. The PPE creates and initializes the matrices, sends them to the SPE and waits for the results. The SPE receive the matrixes, performs the computation and sends back the results and performance to the PPE. Finally, the PPE verifies the results and prints out the performance reached. The matrix sizes and the number of SPEs to use are configurable by command line parameters.

```
int main(int argc, char *argv[]) {
    evaluate_args (argc, argv);
    allocate_and_init_matrixes ();
    for all num_SPEs_used {
        create_new_execution_context ();
        load_spe_matrix_multiplication_code ();
        create_thread_and_start_execution_on_SPE ();
    }
    send_data_to_all_threads ();
    wait_for_all_threads ();
    receive_performance_from_all_threads ();
    print_performance ();
    return 0;
}
```

```
#pragma spe
double spe_matrix_multiplication () {
    receive_data_from_ppe ();
    calculate_matrix_multiplication ();
    send_results_to_ppe ();
    return performance
}
```

Figure 1: Structure of the original matrix multiplication code. The top box corresponds to the PPE code and the bottom box, to the SPE code.

3.2 HELF Library

The basic function of the HELF Library is to allow the programmer to manage the application execution and its behaviour, using the new system calls we have implemented, so that the thread that calls our library will itself execute the function given, in a different core.

The library is fully compatible with other libraries such as Pthreads or OpenMP, so it offers parallelism support as well as heterogeneity.

When an executing thread needs to jump from one execution unit to another, some data and code management must be done in order to prepare the execution environment in the second unit (code and data must be loaded into the unit's local memory). The *helf_execute()* function manages these tasks following the execution sequentially across the different ISAs. Figure 2 shows modifications in the matrix multiplication example to use our library and execute the matrix calculation in the SPE units.

```
int main(int argc, char *argv[]) {
    evaluate_args (argc, argv);
    allocate_and_init_matrixes ();
    for all num_SPEs_used {
        helf_execute (&spe_matrix_multiplication, NULL, &performance, SPE);
    }
    send_data_to_all_threads ();
    wait_for_all_threads ();
    receive_performance_from_all_threads ();
    print_performance ();
    return 0;
}
```

Figure 2: HELF library call adaptation of the matrix multiplication code to calculate the multiplication in the SPE units. The SPE function must call the *helf_exit()* library function at the end, to return the execution, as explained in Table 1.

Similarly, some OS management must be done to return from the executed function. This is done by the *helf_exit()* function.

Table 1 shows the specification of the new functions and their description.

Function name	**helf_execute**
Parameters	- Pointer to the function to be executed. - Pointer to a structure containing the parameters of the function. - Pointer to the address where the value returned by the function must be stored. - Integer representing the architecture where the function must be executed.
Description	This function must be called when a thread wants to jump from one execution unit to another which does not understand the ISA of the first.

Function name	**helf_exit**
Parameters	- Pointer to the address where the return value is stored. - Integer representing the architecture where the function has been executed.
Description	This function must be called when the thread wants to return to the point where helf_execute() was invoked. It is usually called at the end of the function that has been executed by helf_execute().

Table 1: Description of *helf_execute()* and *helf_exit()* functions

3.3 Compiler and linker

The compiler must be adapted to be able to separate the code specified by the programmer inside the *#pragma* directive into the different sections. In the case of using a library, this compiler work would be avoided.

The linker must group all the object files generated by the compiler in a single binary file (there can be more than one object file, depending on the number of source code files and the way they are compiled) and resolve code symbols.

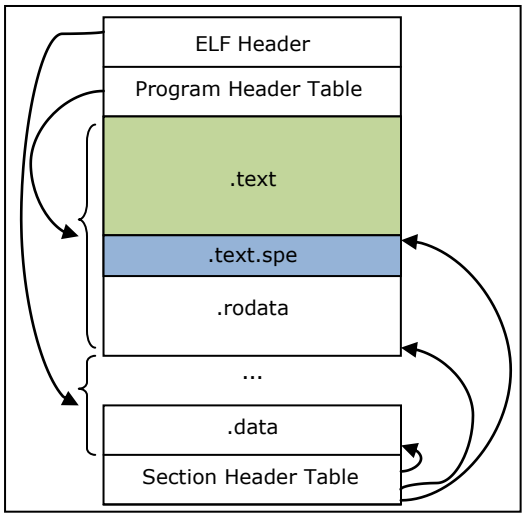

Figure 3: HELF structure of the example

Following the matrix multiplication example, the HELF structure would look like Figure 3: the functions are divided into two different sections: *.text* and *.text.spe*. The name of text sections created starts with "*.text*" followed by a string which is extracted from the *#pragma* indications (see Figure 1), with a "." between them. We use this name to specify the ISA of the compiled code. The main section (*.text*) will contain the main function and all other functions that are not marked with the *#pragma* directive.

If we executed this binary in a conventional operating system we would obtain the same behaviour and would gain nothing from this division because the ELF loader treats all text sections equally: we need a loader able to manage this kind of binary and this is explained in section 4.1.

We assume that these first three steps (source code marks, compiler and linker) are already done and this is our starting point. After that, some operating system modifications are necessary in order to:

- Allow the introduction of information in the operating system per-process data structures
- Maintain information consistency among all the system processes

4. THE OS SIDE

In this section, we will explain the modifications at operating system level to identify and treat the additional information kept in this new executable format.

4.1 HELF Loader

When the file we want to execute is loaded, the part of the HELF header which contains the magic number is compared with the new magic number. If they are different (i.e. the file isn't a HELF), the operation finishes its execution, returning an error. If it is a HELF, it continues with the binary loading.

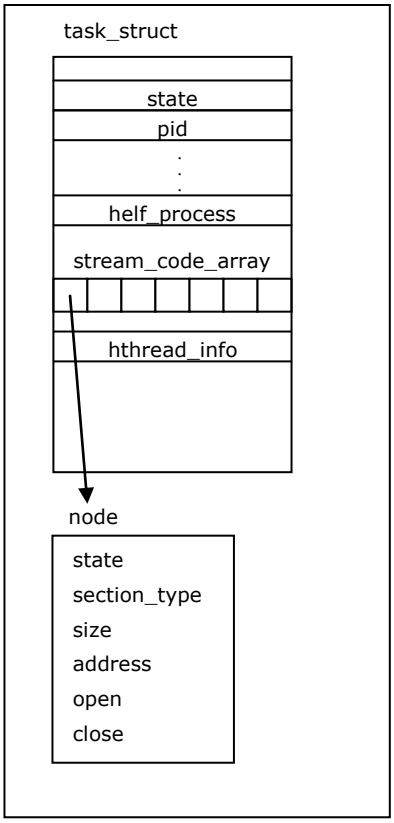

Figure 4: New *task_struct* to represent a heterogeneous process with detail of a node structure.

The *task_struct* structure now has three new fields in order to manage and store the additional information related to HELF binaries (shown in Figure 4):

- *helf_process* indicates whether the process represented by this *task_struct* is a HELF process or not.
- *stream_code_array* contains an array of nodes. Each node structure represents a different section of the HELF binary and contains some useful information, such as the kind of section it represents, its size, its starting address and pointers to operations for managing it. The array size should be sufficient to meet all system needs.
- *hthread_info* contains all the information necessary to maintain the state of the thread before jumping to another execution unit (basically, CPU registers).

Values for a *task_struct* corresponding to a non HELF process would be zero for the *helf_process* field and invalid for each element of the array and the *hthread_info*.

Two important fields in the node are the pointers to the operations that perform the jump between execution units. These operations are architecture-specific. Depending on the ISA section, the loader initializes these pointers to the specific ISA functions.

Following the matrix multiplication example presented previously, the structures created would be filled with the information contained in the HELF binary. The field *helf_process* would value 1 (this is a HELF process), and the first position in the *stream_code_array* would contain pointers to operations for managing the section *.text.spe*.

4.2 Execution

In the previous section we explained the interface offered by the HELF user library, and how the applications must be adapted to fit our model. In this subsection we focus on the back interaction between the library and the operating system.

The two library functions presented invoke two system calls. These system calls will search for the node that represents the architecture where we want to jump to or return to. If this node exists, the open or close functions stored in the node fields will respectively be invoked.

This method is similar to that of the Linux device driver, where a structure stores all the operations that manage each device.

Figure 5 shows the different steps followed by the operating system when an application calls the HELF Library and its effects from the application point of view. Each colour represents the execution unit of a different ISA.

OS control functions are executed in the general purpose elements, so, from the point of view of the execution element (or thread), we can say that heterogeneity always performs in a sequential way: the main processor forks in a new thread (by means of OpenMP, Pthreads or any other thread library) and this new thread is responsible for loading and executing code in a different engine).

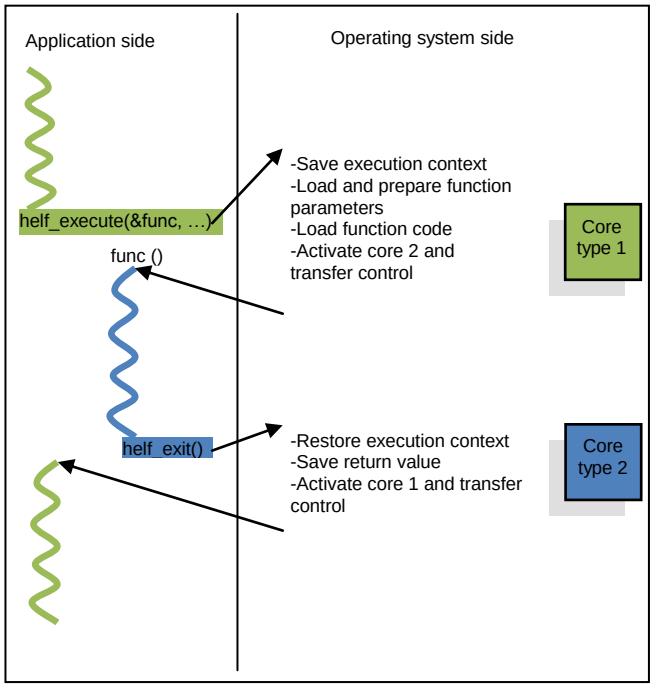

Figure 5: Cross-ISA execution flow of an application using the HELF Library to jump from one core to another.

The system provides some information related to every HELF process that is being executed. In particular, information about HELF sections in memory for each process can be read through the /proc file system. Figure 6 shows the information extracted from /proc for the program in Figure 1.

```
$> cat /proc/helf
Information about HELF processes:
PID: 22266
  HELF Sections: 1
  Section #1:
    type: ppe
    size: 0x73d B
    address: 0x08048a9c
```

Figure 6: Information read from /proc/helf when an application is running on the system.

5. MEASUREMENTS AND RESULTS

In this section we explain the tests we have run. At first, our project was only implemented for the x86 architecture. But as the project went on, we started migrating our implementation to the Cell BEA. Currently, our proposal can run on both architectures, so we have measured it on these test platforms:

- Intel Core 2 Duo processor at 1.86 GHz, with 2 GB of memory.
- PlayStation 3 Cell BE.

Both platforms use our modified kernel, based on version 2.6.24.3. The operating system is Fedora Core 8.

We have performed four different tests to evaluate different aspects of our proposal:

- Measured the binary loading time to evaluate the overhead introduced at kernel level.
- Counted the increment of source lines of code (SLOC) in the kernel, to be able to judge its portability to other platforms.
- Tested the Scimark2 benchmark in the Intel Core 2 Duo and the PowerPC (the PPE part of the Cell BE) architectures.
- Tested an implementation of a matrix multiplication in the Cell BE architecture, using both the PPE and the SPEs.

5.1 Measuring the kernel overhead

The first part of the tests consists of measuring the overhead introduced at kernel level. We generate three versions of the same application:

- ELF: The application with no changes (standard compilation and execution).
- HELF 0: The binary's magic number is changed, so the operating system treats it as a different kind of binary, but the information contained and its structure is the same as the ELF version. We call it HELF 0 because it has no information about heterogeneity.
- HELF 6: This version has six new sections, representing different ISAs. The loader will find extra information and it will have to save it in the process' *task_struct*.

Taking the ELF version as the base, there is no overhead introduced for the HELF 0 version tests and a few microseconds for the HELF 6 version tests, which is negligible.

5.2 Source lines of code

The second thing we want to measure is the number of source lines of code (SLOC) we have added to the Linux kernel.

The approximate number of source lines of code added is detailed below:

- HELF Loader: 750 lines.
- Specific architecture functions: 100 lines.
- Auxiliary functions invoked in different kernel functions: 50 lines.

- New system calls: 60 lines.
- Data structure and constants definitions: 50 lines.

As we can see, most of the new lines belong to the new loader (75%). Approximately, the 2.6 Linux kernel has 5.2 million SLOC, so our proposal represents an increment of 0.02%.

5.3 Scimark2

In this section we present the tests we have made, based on the Scimark2 benchmark[5] for two different homogeneous architectures: x86 and PowerPC (the Cell BE PPE element).

Firstly, we compared the size (in bytes) of the files between the original benchmark and the adapted version. In particular, we compare the object files generated at compilation time, as well as the executable file. The results are shown in Table 2.

File	PowerPC			x86		
	Original	HELF	%	Original	HELF	%
FFT.o	4020	4020	0	2568	2568	0
LU.o	2332	2332	0	1564	1564	0
Montecarlo.o	1624	1624	0	1192	1192	0
Random.o	3628	3628	0	2296	2296	0
SOR.o	1764	1764	0	1188	1188	0
SparseCompRow.o	1440	1440	0	943	943	0
Stopwatch.o	2136	2136	0	1504	1504	0
array.o	1928	1928	0	1452	1452	0
kernel.o	5008	5844	+17	3956	4372	+11
scimark2.o	4444	5492	+23	3704	4240	+14
scimark2	26799	28000	+5	16369	17189	+5

Table 2: Comparison of the file sizes between the original and the adapted compilation for both architectures: PowerPC and x86.

As we can see in the table, the size of the executable file (scimark2) is increased by only a 5%, which we consider acceptable, and the size of its object file (scimark2.o) is increased considerably. The size of the kernel.o object is also increased slightly, and the size of the other files is unchanged, as there is no need to modify them in order to adapt the application to our project.

Secondly, we measured the application performance. The application itself gives us information about its throughput (MFlops). The benchmark provides two different options: small cache-contained data structures and large data structures (which typically do not fit in cache). In these tests we were not concerned about the runtime memory usage or management. That is why we have chosen the small cache-contained version. We test a sequential and a parallel execution (using the Pthread library).

The throughput results for the sequential execution are shown in Table 3, and the execution times are shown in Chart 1.

Function	PowerPC		x86	
	Original	HELF	Original	HELF
FFT	28,86	29,07	138,87	138,43
SOR	108,11	108,21	474,62	465,45
MonteCarlo	9,28	9,33	46,60	46,74
Sparse matmult	34,47	34,92	190,18	190,62
LU	47,49	47,44	249,52	250,98

Table 3: Throughput results for the sequential execution obtained with the original and the adapted benchmark for both architectures: PowerPC and x86.

Although the overall throughputs are higher in the HELF execution, the differences are generally insignificant, except in the case of the SOR function.

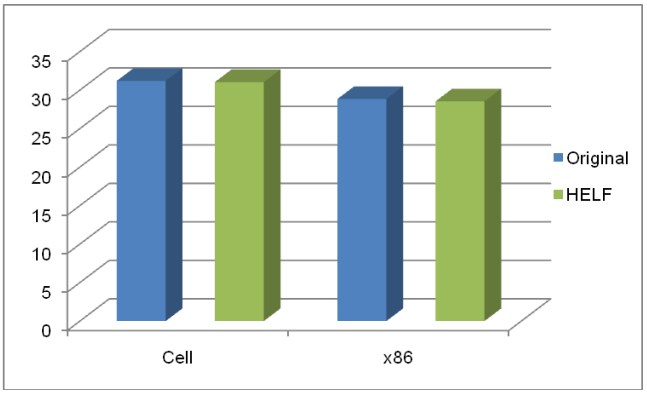

Chart 1: Sequential execution time (in seconds) obtained with the original and the adapted benchmark for both architectures: PowerPC (Cell PPE unit) and x86.

The execution in the x86 architecture is slightly faster (this may be due to the different amount of memory), and both HELF benchmarks run a few seconds faster than the original versions.

The throughput results for parallel execution are shown in Table 4, and the execution times are shown in Chart 2.

[5] This benchmark can be found at http://math.nist.gov/scimark2/

Function	PowerPC		x86	
	Original	HELF	Original	HELF
FFT	4,60	4,10	38,08	30,36
SOR	20,61	22,51	99,09	108,86
MonteCarlo	1,42	1,62	12,12	14,12
Sparse matmult	7,47	6,86	37,83	56,99
LU	7,62	7,83	73,01	68,44

Table 4: Throughput results for the parallel execution obtained with the original and the adapted benchmark for both architectures: PowerPC and x86.

As with the sequential execution, the overall throughput obtained is greater in the case of the HELF benchmark in both architectures, except in some particular cases where the original throughput is a little higher. In this case, the throughput differences are greater than those obtained in the sequential execution.

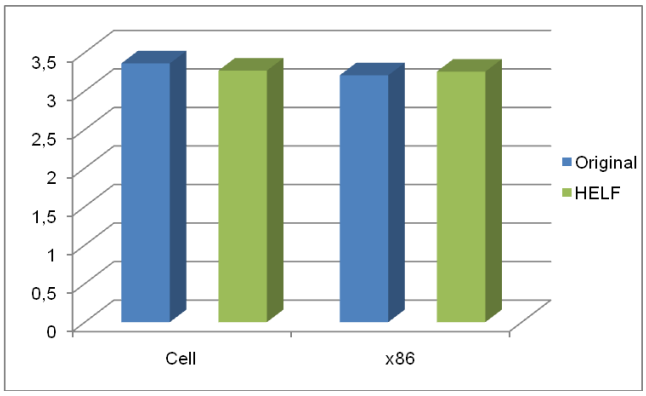

Chart 2: Parallel execution time (in seconds) obtained with the original and the adapted benchmark for both architectures: PowerPC and x86.

In the parallel execution case, the execution times are very similar and much lower than the sequential execution (as happens in many applications when they are parallelised). The time execution of the HELF version in the Cell architecture is lower than that of the original version and the opposite case happens in the x86 architecture. Even though, the time difference between the HELF and original benchmarks is greater in the case of the Cell architecture.

5.4 Matrix multiplication on the Cell BE

This last test is performed in the Cell BE platform, using both PPE and SPE ISAs. We measured the performance (this matrix multiplication calculates its throughput in GFlops) and time consumption of the original application (a simple matrix multiplication, compiled as an ELF binary) and an adapted version using the HELF library (compiled as a HELF binary).

The matrices size is 3200x3200 floats, and the application was tested using one SPE and four SPEs.

- The throughput results for the execution using one SPE are exactly the same in the original version and in the HELF version (25,37 GFlops of 25,60 GFlops theoretical peak).
- Like the previous test, the throughput using four SPEs is also the same in the original and in the HELF version (100,97 GFlops of 102,40 GFlops theorical peak).

The execution times for both tests are shown in Chart 3.

As we can see in the chart below, the execution time of the HELF versions is slightly longer than that of the original versions (a few milliseconds), because the operating system side is not completely implemented and we use parts of libspe library, combined with our implementation.

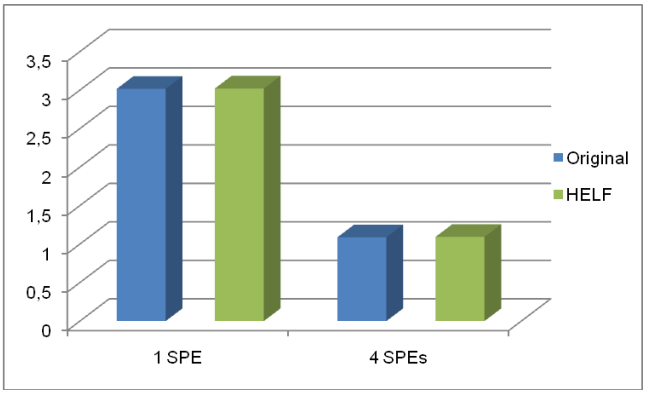

Chart 3: Execution time (in seconds) obtained with the original and the adapted matrix multiplication using one and four SPEs.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present a new method for managing heterogeneous binaries. We have based our proposal on the continuation model of the Mach microkernel and the device drivers of Unix-Linux system.

At application level, we modify the application's source code by inserting new information about the heterogeneity of its different code parts. We have implemented a library that helps the programmer to manage the application execution, and allows a thread to continue execution of different functions in different cores, jumping from one to another.

Heterogeneity is orthogonal with parallelism, so that our library is fully compatible with other libraries such as Pthreads, OpenMP or libspe.

At operating system level, we add the ability to recognize new heterogeneous binary extensions and fill each process' *task_struct* with the related information in order to schedule it in the appropriate engine to achieve the optimal performance. We have implemented the architecture-

specific operations which perform the thread jump between different cores.

We have tested our project at kernel level as well as user level with optimistic results. The tests have been performed in general purpose processors (the implementation is just a first approach and by this way the tests are simpler. The most important result is that the overhead introduced is negligible, so we might expect a better performance when executing the application in the appropriate kind of accelerators.

As future work, more profiling information is needed and different resource management policies should be tested. Finally, a deeper study and evaluation about using different heterogeneous platforms such as GPGPUs or FPGAs is also part of the ongoing work.

7. ACKNOWLEDGMENTS

We sincerely thank Lluc Alvarez for helping us in adapting our proposal to the Cell architecture and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the BSC (Barcelona Supercomputing Centre).

8. REFERENCES

[1] Bellens, P., Perez, J. M., Badia, R. M. and Labarta, J. 2006. *CellSs: a Programming Model for the Cell BE Architecture*. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing.

[2] Chow, A. 2006. *Programming the Cell Broadband Engine*. In Embedded Systems Design magazine.

[3] Draves, R. P., Bershad, B. N., Rashid, R.F., Dean, R.W. 1991. *Using Continuations to Implement Thread Management and Communication in Operating Systems*. Technical Report CMU-CS-91-115, Carnegie Mellon University. Also appears in Proceedings of the Thirteenth Symposium on Operating Systems (SOSP).

[4] Eichenberger, A. E., O'Brien, J. K., O'Brien, K. M., Wu, P., Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M. K., Archambault, R., Gao, Y. and Koo, R. 2006. *Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture*. IBM System Journals, vol. 45, no. 1.

[5] Gil, M., Alvarez, L., Joglar, X., Planas, J. and Martorell, X. 2007. *Operating System Support for Heterogeneous Multicore Architectures*. In UPC-DAC-RR-CAP-2007-40.

[6] Gschwind, M., Erb, D., Manning, S. and Nutter, M. 2007. *An Open Source Environment for Cell Broadband Engine System Software*. In Computer, vol. 40, no. 6, pp. 37-47.

[7] Hester, P. 2007. *Multicore and Beyond: Evolving the x86 Architecture*. In Symposium on High Performance Chips.

[8] Hunt, G. C., Larus, J. R., Tarditi, D. and Wobber, T. 2005. *Broad New OS Research: Challenges and Opportunities*. In Proceedings of Tenth Workshop on Hot Topics in Operating Systems, Santa Fe, N.M.

[9] Sondag, T., Krishnamurthy, V. and Rajan, H. 2007. *Predictive Thread-to-Core Assignment on a Heterogeneous Multi-core Processor*. In Fourth Workshop on Programming Languages and Operating Systems (PLOS 2007).

[10] Tool Interface Standards (TIS) Committee. 1995. *Executable and Linking Format (ELF) Specification, version 1.2.*

[11] Wang, P., Collins, J., Chinya, G., Jiang, H., Tian, X., Girkar, M., Pearce, L., Lueh, G., Yakoushkin, S. and Wang, H. 2007. *Accelerator exoskeleton*. Intel Technology Journal, vol. 11, issue 03.