D4.1– Preliminary report on runtime optimization and tuning Version 1.0

Document Information

Contract Number	610402
Project Website	www.montblanc-project.eu
Contractual Deadline	MM12
Dissemination Level	Public
Nature	Report
Authors	Chris Adeniyi-Jones (ARM), Xavier Martorell (BSC)
Contributors	Roxana Rusitoru (ARM), Regina Lio (ARM), Javier Bueno (BSC)
Reviewers	Axel Auweter (BADW-LRZ)
Keywords	HPC, Linux, Runtime

Notices: *The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 610402.*

Change Log

Version	Description of Change
v0.1	Initial draft released to internal reviewers
v1.0	Version released to the European Commission

Contents

Executive Summary

This report describes work done in three areas relevant to the performance of the Mont-Blanc prototype system.

The first section looks at handoptimizing HPC applications to make better use of the NEON instruction set implemented in both the ARMv7-A and ARMv8-A architectures. While gcc successfully vectorizes simple kernels, it fails to automate other common optimization strategies (such as reordering nested loops) and some real-world applications benefit more from revised algorithms than vectorization. Handoptimization should become less necessary as the compilers for ARMv8-A mature but it is important to understand potential performance uplift.

The second section reports on a study into the effect of using transparent huge pages when running HPC proxy applications on ARMv7-A platforms. On an artificial memory access latency benchmark, huge pages show a substantial improvement. Yet, the performance of our application benchmarks remained unaffected by huge pages, at least when using the small datasets that the current evaluation boards permit us to run.

The third section presents the current status of the porting of the OmpSs@cluster runtime system, and preliminary results obtained in the matrix multiplication benchmark.

1 Optimization of HPC mini apps and Mont-Blanc microbenchmarks

1.1 Introduction

The aim of this study was to investigate the performance of typical HPC applications on ARM and to provide an optimized implementation of a few micro-benchmarks and HPC mini applications on the ARMv7-A and more importantly the ARMv8-A architectures.

In this work, optimization techniques such as vectorization with NEON intrinsics, software pipelining, memory alignment, rearranging data access patterns were explored. We evaluated the performance and cost of hand optimizing these applications against the performance obtained by compiler optimization, and hence assessed the current level of compiler auto-vectorization. We focused on optimizing applications on a single core, thus all applications were run with their serial implementation.

Two kernels, vector operation and dense matrix multiplication, were chosen from the WP6 microbenchmarks suite for optimization. HPC mini applications CoMD [1] and HPCG [2] were also studied.

1.2 Methodology

1.2.1 Hardware setup

In this project, the Mont-Blanc WP6 benchmarks were optimized for both ARMv7-A and ARMv8-A AArch64 on the Arndale board and the Juno board respectively. CoMD and HPCG were only optimized for ARMv8-A AArch64 on the Juno board. Configurations of the boards are as follow,

Arndale Board

- Version 2.0 Samsung Exynos 5 Dual
- 1.7 GHz dual-ARM Cortex-A15 (Caches: L1 32KB I 2-way set associative, 32KB D 2-way set associative, L2 1MB 16-way set associative)
- Set to performance governor (1.7GHz)

Juno Board

- Dual Cluster, big.LITTLE configuration
- Cortex-A57 MP2 cluster (Overdrive 1.1GHz operating speed, Caches: L1 48KB I 3-way set associative, 32KB D 2-way set associative, L2 2MB 16-way set associative)
- Cortex-A53 MP4 cluster (Overdrive 850MHz operating speed, Caches: L1 32KB, L2 1MB)
- Taskset to a Cortex-A57 at 1.1GHz

1.2.2 Compilers

All default performance numbers are obtained by compiling the original code with flags that yielded the best performance.

ARMv7-A compiler

arm-linux-gnueabihf-gcc (crosstool-NG linaro-1.13.1-4.8-2013.10 - Linaro GCC 2013.10) 4.8.2 20131014 (prerelease)

ARMv7-A compilation flags

Set 1: `-O3 -Wall -g`

Set 2: `-O3 -Wall -g -fomit-frame-pointer -funroll-loops -mcpu=cortex-a15 -mtune=cortex-a15 -mfloat-abi=hard -mfpu=neon -funsafe-math-optimizations -ftree-vectorize -DL2SIZE=1048576`

ARMv8-A compiler

aarch64-linux-gnu-gcc (crosstool-NG linaro-1.13.1-4.9-2014.06-02 - Linaro GCC 4.9-2014.06) 4.9.1 20140529 (prerelease)

ARMv8-A compilation flags

Set 1: `-O3 -Wall -g`

Set 2: `-O3 -Wall -g -march=armv8-a+fp+simd -ftree-vectorize -fomit-frame-pointer -funsafe-math-optimizations -funroll-loops`

Set 3: `-std=c99 -g -O3 -static -I. -Wall -march=armv8-a+fp+simd -ftree-vectorize -funroll-loops -fomit-frame-pointer`

Set 4: `-O3 -g -march=armv8-a+fp+simd -ffast-math -ftree-vectorize`

1.2.3 Parameters

All benchmarks were ran with the default parameters listed as follows.

vector-operation: numElements 16777216

dense matrix multiplication: dimension 2048

CoMD: 32000 atoms, 20x20x20 lattice

HPCG: problem size 104x104x104, execution time 60s

1.2.4 Software

To aid optimization decisions, code development and debug, *Streamline* [3] and *gdb* [4] were used. However, the level of functionality supported by *Streamline* for the Juno board was not enough to enable optimization decisions based on Streamline analysis.

1.3 Vector operation

This benchmark adds two vectors together and stores the result in a third vector. Due to its simplicity we expect the compiler to be able to fully vectorize the kernel.

1.3.1 Implementation and Optimization

The original implementation consist of one `for` loop that accesses each element in the two input arrays in order, and store their sum in a third array. Optimizations performed on this micro-benchmark include adding NEON intrinsics, software pipelining (loop unroll 2x), and experimentation with `__restrict` pointers along with relative indexing.

1.3.2 Performance and Analysis

As shown in Table 1, 2 and 3, performance of the hand-optimized versions are fairly similar to the compiler optimized versions of the code. This is as expected since vector-operation is a very simple benchmark and should be easy for the compiler to auto-vectorize the loop. The small differences in performance across the versions are not statistically significant given the uncertainty in performance measurement.

Function call	**Description**	**Flags**	**Relative performance**
addOperationKernel	Original	Set 1	1
addOperationNeon	NEON intrinsics	Set 2	1.03
addOperationNeon1_2	NEON intrinsics, loop unroll, software pipellining	Set 2	1.02
addOperationNeon2	NEON intrinsics, const __restrict ptr, relative indexing	Set 2	1.02

Table 1: Performance of vector operation on Arndale ARMv7-A float

Function call	Description	Flags	Relative performance
addOperationKernel	Original	Set 2	1
addOperationNeon	NEON intrinsics	Set 2	0.98
addOperationNeon1_2	NEON intrinsics, software pipellining	Set 2	0.97
addOperationNeon2	NEON intrinsics, const __restrict ptr, relative indexing	Set 2	0.96

Table 2: Performance of vector operation on Juno ARMv8-A float

Function call	Description	Flags	Relative performance
addOperationKernel	Original	Set 2	1
addOperationNeonDouble	NEON intrinsics	Set 2	1.02
addOperationNeonDouble	NEON intrinsics, memory alignment (32 byte aligned)	Set 2	1.03

Table 3: Performance of vector operation on Juno ARMv8-A double

1.4 Dense matrix multiplication

This benchmark multiplies two dense matrices together and stores the result in a third matrix. It is characterised by regular access patterns. The reference code allows for treating the matrices as either row-major or column-major matrices. Our optimized versions treat the matrices as row-major matrices.

1.4.1 Implementation, Optimization and Performance Analysis

The original version of dense matrix multiplication employs tiling which avoids data being evicted from cache before it is needed again. To optimize this kernel, we first performed memory alignment. From experiments, we found that aligning data to 16 byte gave the best performance, resulting in a 67% speedup.

We then incorporated NEON intrinsics to the matrix multiplication kernel in a similar manner as described in the NEON programmers guide[5] to work on sub-blocks of the matrices. Thirdly, a loop interchange was performed to reorder execution of the nested loops exploiting spatial locality of data in the cache. This simple loop interchange resulted in the greatest performance improvement on all three configurations. Other loop interchange and software pipelining were also investigated.

To further exploit temporal locality in the cache, a *double blocking* method was devised. This configuration essentially treats each NEON 4x4 float/2x2 double block as one matrix element and performs a second layer of blocking and loop interchange in a similar manner as mentioned previously.

The performance of all optimizations is presented in Table 4, 5 and 6. All versions are compiled with Set 2 flags unless otherwise stated. Where Set 2 flags are not used it is because they caused an internal compiler error during compilation.

It should be noted that there are two NEON intrinsic instructions (`vfma` and `vmla`) are very similar in functionality, performing a fused multiply-add and a multiply-add operation respectively. Both of these instructions have been experimented with in this project. From Table 5, we can see replacing `vmla` with `vfma` reduced performance slightly. However, in some cases [1], using `vmla` did not produce a valid result due to its rounding nature so one must consider their accuracy requirements before deciding on either operator.

Function call	**Description**	**Relative performance**
baseMatrixMultiplication-FuncRowMajor	Original	1
denseMatrixMult_Neon_-v1_RowMajor	NEON intrinsics float 4x4 vectorized block (mla)	1.7
denseMatrixMult_Neon_-v2_RowMajor	As above with loop interchange i, k, j (mla)	7.3
denseMatrixMult_Neon_-v3_RowMajor	As above with loop interchange k, i, j (mla)	7.5

Table 4: Performance of dense matrix multiplication on Arndale ARMv7-A float

[1] running `denseMatrixMult_Neonv1_RowMajor` on Juno ARMv8 float did not validate

Function call	Description	Relative performance
baseMatrixMultiplication-FuncRowMajor	Original[1]	1
denseMatrixMult_Neon_-v5_RowMajor	NEON intrinsics float 4x4 vectorized block (fma)	1.2
denseMatrixMult_Neon_-v2_RowMajor	As above with loop interchange i, k, j (mla)	4.9
denseMatrixMult_Neon_-v3_RowMajor	As above with loop interchange k, i, j (mla)	4.8
denseMatrixMult_Neon_-v4_RowMajor	As above with loop interchange k, i, j (fma)	4.2

[1] Compiled with Set 1 flags

Table 5: Performance of dense matrix multiplication on Juno ARMv8-A float

Function call	Description	Relative performance
baseMatrixMultiplication-FuncRowMajor	Original[1]	1
baseMatrixMultiplication-FuncRowMajor	Original, memory alignment (16 byte aligned)	1.67
denseMatrixMult_Neon_v1-_RowMajor_Double	NEON intrinsics double 2x2 vectorized block (fma)	9.1
denseMatrixMult_Neon_v2-_RowMajor_Double	As above with loop interchange k, i, j	6.1
denseMatrixMult_Neon_v3-_RowMajor_Double	Software pipeline, Loop unroll to 4x4 vectorized block, i, k, j	6.4
denseMatrixMult_Neon_v4-_RowMajor_Double	2x2 vectorized block with outer block 128x128	9.6

[1] Compiled with Set 1 flags

Table 6: Performance of dense matrix multiplication on Juno ARMv8-A double

1.5 CoMD

The molecular dynamics computer simulation method is often used for the study of thermodynamic properties of liquids and solids. CoMD is a molecular dynamics proxy application featuring the Lennard-Jones potential (ljForce) and the Embedded Atom Method potential (eamForce). In this project we only studied the ljForce kernel as the eamForce kernel builds on top of it. This implementation (Version 1.1) considers materials with short range inter-atomic potential (e.g. uncharged metallic materials) and encapsulates only the most important computational operations, the evaluation of the force acting on each atom due to all other atoms in the system and the numerical integration of the Newtonian equations. [1]

1.5.1 Implementation, Optimization and Performance Analysis

The original implementation uses a well-researched link cell method. This method is effective when the number of atoms is large.[6] The work region is decomposed into boxes with width defined by the interaction cut-off radius r_{cut} of an atom. All atoms are assigned to these boxes by their positions. With the width of the box equals to r_{cut}, one only have to search though the 27 neighboring boxes (including the one it resides) to collect all the neighbors an atom can possibly interact with, eliminating the need to search the entire work space.

Listing 1: Psuedocode of ljForce original implementation

```
for all iBox in localboxes do
  if numberOfAtoms(iBox) is 0 then exit loop; end if;
  for all jBox in getNeighbourBoxes(iBox) do
    if numberOfAtoms(jBox) is 0 then exit loop; end if;
    for all iAtom in iBox do
      for all jAtom in jBox do
     // avoid double counting atom pair interaction
     if jAtom.id < iAtom.id then exit loop; end if;

       calculate separation of iAtom and jAtom;
     if separation > cutoffRadius then exit loop; end if;

     calculate PotentialEnergy;
     calculate Force;
        update iAtom, jAtom, totalPotentialEnergy;
      end for;
    end for;
  end for;
end for;
```

The force kernel (Code Listing 1) which evaluates the potential energy of the system, comprises 99% of the computational effort in CoMD. Both ljForce and eamForce function loop over pairs of atoms to determine their separation, and hence whether the atoms interact with each other. Since the force exerted upon each atom in a pair is equal, we only need to compute a force once for each pair of atoms (rather than once for each atom in the pair). The ljForce kernel consists of nested `for` loops with `if` statements in the inner loops which introduce control flow divergence, making it difficult to fully vectorize the operations.

Consequently, introducing NEON intrinsics and subsequent fall back code for odd atom pairs did not introduce much performance improvement, namely a mere 1% speedup

which is highly likely to be noise in the measurements. A first attempt in overcoming this problem was to collect atom pairs at the first main control flow divergent point, where atom pairs were disregarded because the force between them has already been computed once. This allowed us to identify the separation of two atom pairs at one time, but does not eliminate the need of fall back code at the second main divergent point, where the we exit the loop if the separation between the atom pair is greater than r_{cut}. As a result we only gain a 3% performance improvement in this configuration.

Combined neighbor list and link cell method

The next solution takes inspiration from research work by S. Plimpton [7], G. Grest [8] and Z. Yao et al.[6]. Their work describe various methods of incorporating a neighbor list, originally proposed by Verlet [9][2], with the link cell method. Our implementation is closest to the work by Z. Yao et al.[6].

In essence, the idea of a Verlet neighbor list algorithm is to construct and maintain a list of neighboring atoms for each atom in the system. During the simulation, the neighbor list will be updated periodically in a fixed interval, or automatically when the displacement of two atoms in a box is greater than a skin layer r_{skin}. When constructing the neighbor list of atom i, an atom j is considered as a neighbor if the separation between the two atoms are within $(r_{cut} + r_{skin})$. An important point of this algorithm is that, r_{skin} should be large enough such that no atom can penetrate through the skin into the cutoff sphere of another atom if it is not in the neighbor list of that atom before the next reconstruction of the neighbor list.

Combining Verlet's neighbor list with the existing link cell method means that we update the neighbor list of each atom by searching in the 27 neighboring boxes, or in the volume of $27(r_{cut} + r_{skin})^3$. Afterwards, during each force update iteration, we identify neighboring atom j with separation distance less than r_{cut} from atom i's neighbor list. This represents a search space of the sphere volume $\frac{4}{3}\pi(r_{cut} + r_{skin})^3$, about 22%[3] of the original search space of 27 r_{cut}^3.

Implementing this algorithm requires several modification to the original code described as follows,

- New member `dirtyFlag` in `LinkCellSt` structure. This data member that indicates if it is necessary to reconstruct the neighbor list of all atoms in the box (LinkCell) at the start of an iteration. This dirty flag is updated every iteration and is set to 1 whenever the sum of the displacement of two atoms in a box is greater than r_{skin}. The dirty flag of the resident boxes and neighboring boxes is also set when an atom crosses a link cell boundary. [6].

- New member neighbor list `nbrList` in `AtomSt` structure. This is an array containing pointers to an atom's neighbor that is positioned within the $(r_{cut} + r_{skin})$ sphere.

- New member number of neighboring atoms `nNbrAtoms` in `AtomSt` structure.

[2]Also detailed in Allen and Tildesley's book [10]

[3]For a typical value of $r_{skin} = 0.12r_{cut} = 0.3\sigma$. σ is the finite distance at which the inter-particle potential is zero

- New member `lastR` in `AtomSt` structure. This data member stores the atom's position after every neighbor list reconstruction, it is used to calculate the displacement of an atom since the last neighbor list update.

With $r_{skin} = 0.3\ \sigma$, simulating with the default supplied potential copper, atoms cross link cell boundary at 30+th iterations, resulting in an error in the force calculation[4] that *has yet to be fixed*. If we increase the skin layer thickness to $r_{skin} = 0.34\ \sigma$, box width is now greater. As a result atoms are unlikely to cross the cell boundary for this particular simulation potential.

We expect vectorisation using NEON intrinsics to be more effective for this method as a result of knowing that the neighbour atoms are within range. Indeed the performance improvement using the new algorithm in this case is 1.22x.

The performance for all version are presented in Table 7. All versions are compiled with Set 3 flags.

Version	**Description**	**us/atom/task**	**Relative performance**
serial	Original	10.38	1
neon	NEON intrinsics	10.25	1.01
neon	NEON intrinsics, collect workable atom pairs at divergent point A	10.06	1.03
neon verlet [1]	NEON intrinsics with neighbor list	8.54	1.22

[1] Requires debug

Table 7: Performance of CoMD on Juno ARMv8-A double

1.6 HPCG

HPCG is a synthetic benchmark for ranking computer systems based on a simple additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient solver. HPCG is similar in its purpose to High Performance Linpack (HPL), however, it aims to better represent how today's applications perform. In this project we worked on the latest available version of HPCG, Version 2.4. [2]

Being a synthetic benchmark, there are tight restrictions on permitted transformation and optimizations as detailed in the HPCG Technical Specification [2]. Some notable ones include

- User is not allowed to change algorithms
- User is not allowed to exploit the symmetric properties of the matrix
- Storage format of data can be changed but time is recorded and used in the computation of performance

[4] Discrepancy in final potential energy in the 4th/5th decimal point

- Vector data must be accessed indirectly in the ComputeSpMV and ComputeSYMGS kernels

1.6.1 Implementation, Optimization and Performance Analysis

There are four main floating-point workloads in HPCG, the preconditioner using a symmetric Gauss-Seidel sweep `ComputeSYMGS`; computation of the sparse matrix-vector product `ComputeSpMV`; computation of weighted vector sum `ComputeWAXPBY` and computation of dot products `ComputeDOT`. Out of the four aforementioned kernels, ComputeSYMGS is significantly more difficult to optimize because of its inherent serial nature. The said kernel also constitutes 55% of the total number of FLOPS in the program, promoting it as the critical kernel in HPCG.

The other three kernels ComputeSpMV, ComputeWAXPBY, ComputeDOT were easy to vectorize with NEON. Therefore without surprise, applying NEON intrinsics to these kernels made no difference in performance as the compiler was capable of auto-vectorizing ComputeWAXPBY, ComputeDOT; and the vectorized version of ComputeSpMV still consists of indirect array accesses which is not optimized on our current hardware.

Multiple research papers[11][12][13] have been consulted in the search for a solution to optimize ComputeSYMGS. SYMGS is an iterative method used to solve a linear system of equations. Recurrence of vector elements in the loop implies the algorithm cannot be parallelised easily. One method studied was a multi-coloring SYMGS algorithm adopted by Kumahata et al. working on the K computer[12]. In their work they colored work space that can be operated on in parallel and ran multiple threads on multi-cores using OpenMP. This is not exactly portable to be used on a single core with vector unit like NEON. Hence, being the biggest challenge in HPCG ComputeSYMGS remains unoptimized in this project.

Another main optimization applied by Kumahata et al. was to rearrangement the memory allocated to matrix data such that array memory of matrix values and indices are continuous. A similar approach was taken and this resulted in a performance improvement of 57% due to better spatial locality in the cache. We also performed other optimizations such as combining two ComputeWAXPBY functions to a ComputeTwoMAC function to reduce one loop, and precomputing the reciprocal of the matrix diagonal to transform a division to a multiplication. The gcc `__builtin_prefetch` extension was briefly explored in the dot product kernel but it did not yield any significant performance.

Table 8 shows the performance of the optimized versions of HPCG on an A57 on Juno. All versions are compiled with Set 4 flags.

Description	Relative performance
Original	1
NEON intrinsics	0.99
NEON intrinsics, continuous memory arrangement	1.57
As above with pre-computed reciprocal, combined functions	1.61

Table 8: Performance of HPCG on Juno ARMv8-A double

1.7 Conclusion

In this project, we produced hand-optimized versions of the Mont-Blanc WP6 microbenchmarks, CoMD and HPCG, which can be used as a performance reference for ARMv8 AArch64. A variety of optimization techniques including vectorization with NEON, rearranging memory accesses, loop unrolling, algorithm modification, etc. were exploited to enhance single-core performance of these benchmarks.

We learnt that the latest available gcc compiler are highly capable of auto-vectoising simple kernels like vector-operation, computing dot product and WAXPBY in HPCG. Performance improvement hugely benefit from rearranging memory accesses that exploit spacial and temporal locality in the cache. This is highlighted in the dense matrix multiplication kernel, where we gained performance improvements of up to 9.6x. Likewise in HPCG, adjusting memory allocation ordering contributed a large percentage of the 1.61x speedup.

We were unable to obtain good vectorization on CoMD, but the proposed combined neighbor list and link cell algorithm proves worthy to be revisited in the future.

2 Transparent Huge Pages on ARM Architecture

With systems having increasing amounts of RAM, traditional pages of 4 KiB might no longer enable the system to reach the required performance. With smaller page sizes, it is possible to encounter effects such as TLB thrashing which reduce an application's performance. This is more easily seen in applications with random or sparse memory access patterns. In the Mont-Blanc D5.4 Deliverable - Report on Tuned Linux-ARM kernel [14] we reported on our initial study into the suitability of using Transparent Huge Pages to improve the performance of HPC applications when running under Linux on ARMv7-based platforms. We showed that Transparent HugePages (THP) on the ARM architecture did increase the performance of many of the memory-intensive benchmarks that we executed.
The aim of this section is to show our findings into the effects of using Transparent HugePages on a sample of high-performance computing (HPC) mini-applications when run on the Arndale board using the Samsung Exynos 5 Dual SoC.
We chose to look at hugepages as we believed they can provide a performance improvement for ARM architecture-based systems in the context of HPC applications, due to their effect on the Translation Lookaside Buffer (TLB) and the micro-architecture of the ARM Cortex-A15 L2 TLB, which supports all page sizes for each of its entries. When using larger pages, we observed an increase in L2 TLB hits, better cache usage, and reduced memory latency in the context of applications which have strided or random memory access patterns. Applications with good data locality tend to not show any benefit from using hugepages, as the ARM Cortex-A15 has a hardware prefetcher which can prefetch within a 4 KiB page, irrespective of using smaller or larger pages. However, as soon as we exceed 4 KiB strides, we start encountering TLB thrashing and overall lower performance.
We only look at Transparent HugePages (THP), using 2 MiB pages. Due to the limited virtual address space available on current ARMv7-A-based systems, we do not have support for 1 GiB hugepages.
This work is aimed at evaluating Transparent HugePages for the ARM architecture, and to show that the same overall behaviours are observed on ARM and contemporary architectures. We look at the performance variability between running with and without hugepages.

2.1 ARM architecture and Linux concepts

In this section, we cover some fundamental concepts that can aid in the understanding of later parts of the report.

- ARMv7-A compliant processors can be in one of several modes and states, which determine how the processor operates, including the current execution privilege level and security. The modes a processor can be in are: User mode, System mode, Supervisor mode, Abort mode, Undefined mode, FIQ mode, IRQ mode, Hyp mode, Monitor mode and Secure and Non-secure modes. For further details on these, please visit the ARM Architecture Reference Manual for ARMv7-A [15].

[4] 1 exaFLOP = 1×10^{18} FLOPS

- A page (also referred to memory page) is a block of virtual memory which has the following properties:
  - it is the smallest unit of data that is allocated by the operating system memory allocator for a program;
  - it is backed up by a same-sized physically contiguous region of physical memory;
  - it has a fixed size.
- The Virtual Memory System Architecture of ARMv7-A (VMSAv7), defines two alternative translation table formats:
  - Short-descriptor format: The original format in ARMv7, it is used by all implementations that do not support the Large Physical Address Extension (LPAE). It only provides up to 2 levels of address lookup, with 32-bit input addresses and up to 40-bit output addresses. The tables entries are 32 bits.
  - Long-descriptor format: Added in LPAE, it provides up to 3 levels of address lookup, up to 40-bit input addresses (when they are used for stage 2), and output addresses of up to 40 bits. Table entries are 64 bits.
- Operating System noise is the interference the Operating System has upon an application. Some of the main sources are daemons, services and timer interrupts.
- Translation lookaside buffer (TLB) thrashing occurs when an application constantly requires more memory pages than available TLB entries, thus often causing TLB misses. This has an effect on memory latency, as a hardware page translation needs to occur, which delays the memory accesses.

2.2 Translation table walk for the ARM architecture

Upon performing memory accesses, we require a virtual-to-physical address translation. Modern architectures will typically have one or more TLBs to cache those translations. However, in case we cannot find one a translation table walk occurs. On ARMv7-A, a typical page translation happens in 1 or 2 stages, depending on whether virtualization is being used. When using the Long-descriptor translation table format, the starting level can be either stage 1 or stage 2. In Figure 1 we show how the translation happens for both stages. On ARMv7-A, the address translation is handled by the Memory Management Unit (MMU). If an address translation is cached, the MMU will retrieve it from the TLB, or it is going to perform a table walk. The MMU also controls access permissions, memory attribute determination and checking, for memory accesses made by the processor.

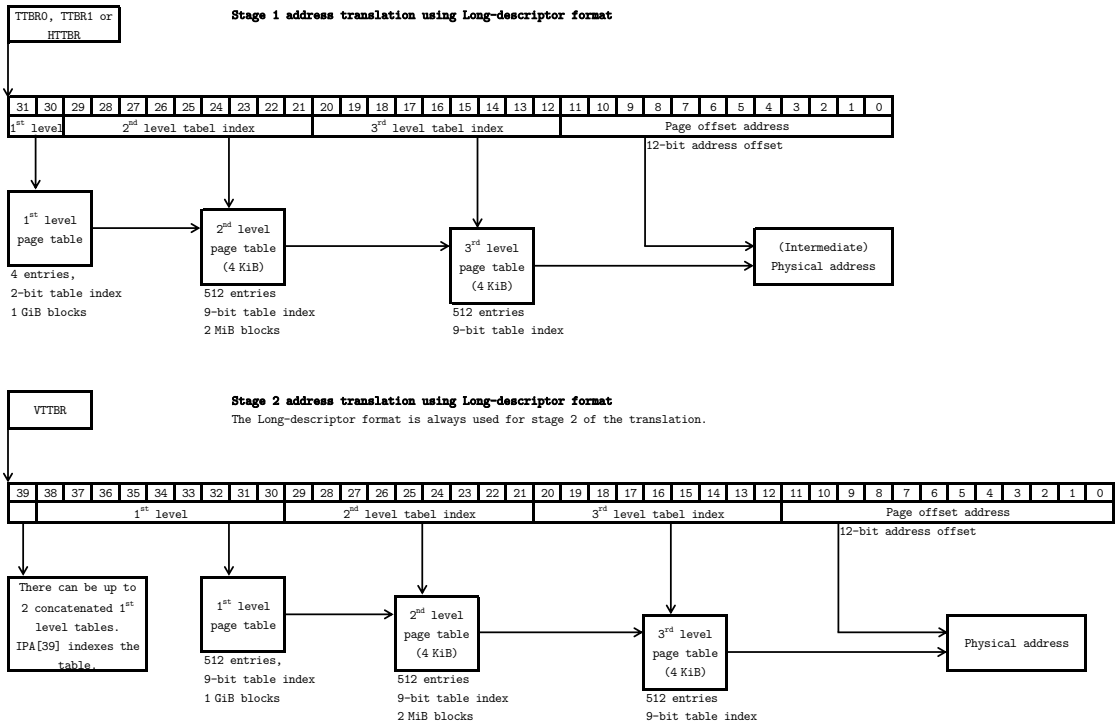

Figure 1: Stages in virtual to physical page translation on ARMv7-A systems using Large Page Address Extension (LPAE)

In the top figure we see stage 1 of the translation from virtual address to either a physical (PA) or an intermediate physical address (IPA), which one obtains if virtualization is used. The Translation Table Base Control Register (TTBCR) determines which of the Translation Table Base Registers (TTBR0 or TTBR1) contains the base address for the stage 1 translation table walk for a memory access from any mode other than Hyp mode. An example would be to use TTBR0 for process-specific addresses, whilst TTBR1 would be used for operating system and I/O addresses that do not change on a context switch. The TTBCR also sets which translation system is used, short or long. The HTTBR points to the stage 1 translation table used for memory accesses from Hyp mode. Depending on which mode the processor is in (Hyp mode or other), we use TTBR0, TTBR1 or HTTBR to go to the correct translation table. Afterwards, we use PA[31:30] to index within the 1st level table, PA[29:21] are used for the 2nd level table, followed by another 9 bits indexing into the 3rd level page table. The final 12 bits are used as the page offset address. When

2 MiB hugepages are in use, only 2 levels of translation are necessary, as the 2nd level maps 2 MiB blocks of memory.
When we use virtualization, then we will have a stage 2 translation as well, as seen in the bottom graph of Figure 1. The process is very similar to stage 1, with a few main differences: the register holding the base address of the translation table is typically the VTTBR, which is equivalent to the HTTBR, TTBR0 and TTBR1 registers, and the input IPA is always 40 bits. In case the address is 32 bits, it is 0 extended to 40 bits. The output of this translation stage is always a physical address.

2.3 Effects of 4 KiB pages on TLB

Translation lookaside buffer (TLB) thrashing typically occurs when there is an insufficient amount of locality in the data accessed. This can affect the performance of many workloads, including HPC ones on a variety of systems and architectures. his issue can easily occur whenever non-linear virtual to physical accesses are used. A way to help with this problem is by using large pages. Under typical circumstances on an ARM Cortex-A15-based system, we have a 512-entry TLB, we are using 4 KiB pages and have a total of 1-4 GiB RAM. This means that if the referenced data crosses the page boundary, only up to 2 MiB can be covered before having TLB misses. However, using a same-size TLB, but 2 MiB hugepages, 1 GiB RAM can be referenced into the TLB, thus significantly decreasing the number of page faults.

2.4 HugePages

As seen in Section 2.3, a way to overcome the limitations of 4 KiB pages is to use larger pages. Depending on the architecture used, pages can be of 16 KiB, 64 KiB or more, with an upper bound of 1 GiB. Some architectures such as PowerPC natively support having the base page size as 16 or 64 KiB, whilst others provide support for using both 4 KiB base pages and hugepages through operating system mechanisms such as *HugeTLB* or *Transparent HugePages*. The former was originally introduced in Linux kernel v2.6 [16], whilst the latter in v2.6.38 [17].
In today's systems where we use gigabytes of RAM, the usual 512-entry TLB can only reference up to 2-32 MiB RAM, depending on what page size is used, which is only a fraction of the total amount of RAM. Given these limitations, the usage of higher sized pages, such as 2 MiB or 1 GiB is preferable.
A previously mentioned method to achieve this is *HugeTLB*, a mechanism which allows the usage of up to 1 GiB pages. This works by reserving a number of contiguous pages to be used by applications through HugeTLB file system (*HugeTLBFS*). With huge pages of these sizes, the number of TLB misses should be at a minimum, however, by using this mechanism we encounter different types of issues. One problem is that, unless we reserve a sufficient amount of RAM at boot time, we risk not being able to find sufficient physically-contiguous space to allocate the required pages. The mechanism can easily suffer from fragmentation if we either don't reserve, or we exceed the amount we put aside initially. Another limitation, which is somewhat compensated by *libhugetlfbs* is the actual usage of the hugepages, which can require code rewrite and/or re-compilation. *libhugetlbfs* allows the user to transparently use hugepages, by replacing the implementation of malloc with its own.
Another, better integrated mechanism is *Transparent HugePages*. It provides a way to let applications use large pages, without making any code changes and no specific user interaction. The way to achieve this is to set it to *always* give a hugepage when possible

after a page fault. The operation is transparent to the user code, and the amount of allocated hugepages is dependent upon the total available RAM. When the system runs out of memory, the mechanism splits a hugepage back into small ones, or only small ones will be allocated after faults. The mechanism makes use of *khugepaged*, a daemon which handles the re-allocation of regular pages to huge pages and defragmentation. A different method to use this is via *madvise* which implies that only `madvise` calls (as seen in Section 2.4.2) get mapped onto hugepages. One can, for example, choose *madvise* over *always* when they want only one particular commonly accessed array or data structure backed up by hugepages, especially when the accesses within this block of memory occur outside of the prefetcher's prefetch stride. It can also be used when we want to ensure that we do not increase the memory footprint of our application.

2.4.1 Latencies

According to 7-cpu.com [18], on ARM Cortex-A15 an L2 TLB miss penalty is estimated at 30 cycles and a page walk to RAM is estimated at around 170 cycles. As we observed in Section 2.3: Effects of 4 KiB pages on TLB, when we have strided or random memory access patterns, using Transparent HugePages can easily provide 30-210% increase in L2 TLB load-store hits. Taking into account the latency of a page walk, we easily see some of the benefits larger pages can provide. We show this in Figure 2, which displays a range of LMbench [19] random memory read runs. We can see that at working set sizes greater than 1 MiB we experience higher contention in the L2 cache, eventually exceeding the cache size and having to go to main memory. At this point, we can see the effects of TLB thrashing on the latency, which increases from around 20-40 ns to 180 ns. Using larger pages improves on this, and saves us the cost of an L2 TLB miss penalty (in this case, around 30 ns). We can also see that using hugepages allows us to get better L2 cache utilization.

Comparing to contemporaries, using a Haswell-based Intel Core-i7, we observe a similar pattern of 2 MiB pages offering better cache utilization and lowering latency when going to main memory.

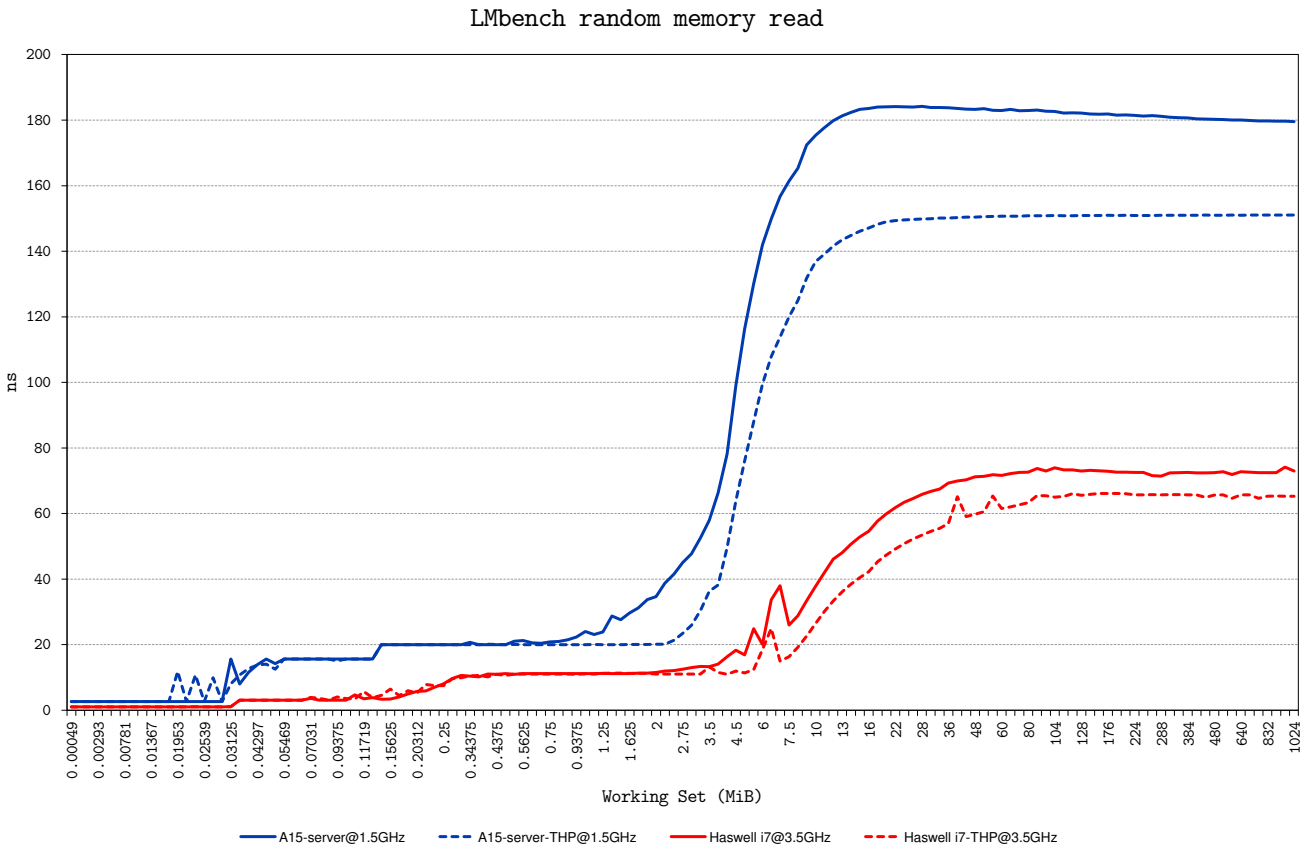

Figure 2: LMbench random memory read latencies[5]

Similar results are described by Milfeld et al. in their paper on Effective Use of Multi-Core Commodity Systems in HPC [20], where they show the latency effects of 2 MiB hugepages on Intel Woodcrest and AMD Opteron systems by using HugeTLB. The former uses a shared unified L2 cache, whilst the latter uses independent unified L2 caches (each core has its own L2 cache). Their benchmark accesses array elements in a strided fashion, and they time how many cycles the operation takes. On the Woodcrest system they show that using larger pages gives them complete L2 coverage, which is other wise limited by the TLB. Their system has a 4 MiB L2 cache and 256-entry TLB, which can cover only up to 1 MiB when using 4 KiB pages. They also observe the sharper transition from the L2 to the main memory latency region and an overall better utilization of the L2 cache. Their analysis on the Opteron system led to similar findings compared to the Woodcrest system. One main difference is that the L2 cache on the Opteron CPU does not suffer from any coverage degradation, as the processor has a 512-entry TLB.

2.4.2 Usage

In order to make use of hugepages, whether through HugeTLB or Transparent HugePages, it is necessary to enable certain kernel configuration options and potentially make code changes. The following sections provide the specific options needed for each mechanism and the most common ways to use them. It is important to remember that with recent kernel versions, typically hugepages are likely to be enabled by default.

[5]Based on data gathered by Eric Van Hensbergen.

HugeTLB

- Kernel config: `CONFIG_HUGETLB_PAGE` and `CONFIG_HUGETLBFS`
- Other tools: In order to use *HugeTLB* transparently, you will need to download the *libhugetlbfs* library [21].
- Usage:
  - In order to use *libhugetlbfs* features, you will first need to mount *HugeTLBFS*. This can be done as:
    ```
    mkdir -p /mnt/hugetlbfs
    mount -t hugetlbfs none /mnt/hugetlbfs
    ```
  - If you wish to use libhugetlb, pass on the command-line the following arguments, followed by the executable to be run with hugepages:
    ```
    LD_PRELOAD=<path-to-libhugetlbfs.so>/libhugetlbfs.so
    HUGETLB_MORECORE=<yes/no/desired-page-size> <application-command-line>
    ```
  - Otherwise, you can use an `mmap` call instead of `malloc` inside your application, as follows:
    ```
    mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS
    | MAP_POPULATE | MAP_HUGETLB, -1, 0);
    ```

Transparent HugePages

- Kernel config: `CONFIG_TRANSPARENT_HUGEPAGE` and `CONFIG_TRANSPARENT_HUGEPAGE_ALWAYS`
- Usage:
  - to enable (does not require code changes), as root:
    ```
    echo "always" > /sys/kernel/mm/transparent_hugepage/enabled
    ```
  - to disable (does not require code changes), as root:
    ```
    echo "never" > /sys/kernel/mm/transparent_hugepage/enabled
    ```
  - There is a 3rd option, called `madvise`, however, it only works if you used `madvise` and marked the virtual memory addresses with `MADV_HUGEPAGE`. Please note that this option is only a hint to the kernel and can be ignored.

2.4.3 Resources

Hugepage support was added in mainline for ARMv7-A with patches `ARM: mm: Transparent huge page support for LPAE systems.` [22] and `ARM: mm: HugeTLB support for LPAE systems.` [23].

2.5 Related work

The usage of hugepages for HPC workloads has been benchmarked in the past, on a variety of systems and applications. Ranjit Noronha [24] used 2 MiB large pages on x86-based systems using HugeTLB to improve the performance of OpenMP applications. The observed improvements were of up to 25 %.
Cox et al. [25] investigated the effects of hugepages on high-end scientific applications whilst running on commodity microprocessors. They investigated the effects of page

sizes whilst using SPEC-fp and HPCC. In an implementation-independent testing environment they observed that when using 2 MiB pages, as opposed to 4 KiB pages, the the number of TLB entries needed to cover 99.9 % of application references were significantly smaller. They would need no more than 512 TLB entries when using large pages, whilst for the 4 KiB pages they would require up to 65536-entry TLBs to achieve the same effect. Notwithstanding, when running sample applications such as *random*, they observed performance degradation when using huge pages that is due to the higher number of TLB misses. This is caused by the fact that the Opteron CPU they used for testing had essentially two separate TLBs: a 512-entry one for 4 KiB pages, and an 8-entry one for 2 MiB pages. As a result, in cases when the application has many random memory accesses, a larger TLB helps, despite the page size. In contrast, the ARM Cortex-A15 has a 512-entry TLB, which supports both 4 KiB and 2 MiB pages sizes [26], thus even when having random memory access patterns, we observe an up to 3× performance improvement when running with THP.

Shmueli et al. [27] experiment with replacing Compute Note Kernel (CNK) for Linux on Blue Gene/L, in order to get a more feature-rich environment. They use a PowerPC PC440-based system, which can only handle TLB misses in software. The system has 2 processors per node sharing 1 GiB of RAM. Each processor has a 64-entry TLB. As a result, they explore the usage of hugepages through *HugeTLBFS* in order to minimize TLB misses. This leads to performance numbers similar to CNK when they map both heap and static data to large pages. Using 16 MiB pages means the Opteron CPU can cover the entire memory region from within their TLB. With our system, the ARM Cortex-A15 can only hold references to 1 GiB RAM, however, due to having a hardware table walk unit, the penalty of a TLB miss is significantly smaller. Furthermore, Large Physical Address Extension (LPAE) systems also support 1 GiB pages, thus for systems with a lot of RAM we could increase the page size further. The results of this whitepaper show the potential of this kernel feature and our results confirm hugepages' suitability for HPC workloads.

In a similar work, Yoshii et al. [28] present their "Big Memory" performance characterization as part of ZeptoOS, a Linux-based operating system aimed at compute nodes. They ran their experiments on a BlueGene machine. Typically BlueGene compute nodes run Compute Node Kernel (CNK), a microkernel which only allows one single user thread per CPU core and provides a very simplistic offset-based physical to virtual memory mapping. This is acceptable, as there is only one user thread, therefore there is no risk of interference and memory corruption. Also, this offset-based mapping is preferred due to the system not having a hardware page walker, therefore all TLB misses have to be handled in software. Nevertheless, this simple design prevents certain desirable general-purpose features such as multi-tasking and time-sharing. Yoshii et al. present an alternative within ZeptoOS, which relies on providing applications with memory regions backed by very large pages of 256 MiB. The memory regions are referred to as "Big Memory". At the time of writing, HugeTLB could only support 2-4 MiB pages, which would reduce the number of TLB misses, whilst their goal is to entirely avoid them. As they are running HPC workloads, they can allow only a single computational process to run on a core at a particular time, thus the kernel can pin down the TLB entries necessary for the Big Memory as the application accesses the memory. All entries are removed when the process gets scheduled out. This ensures the memory mapping to remain private to each process. All "Big Memory" physical memory is reserved at boot time and can only be used by computational processes. The process of requesting larger pages is transparent to the application, such that all allocation requests such as `malloc` get automatically backed by the larger pages, whilst file-backed mappings will end up using small pages.

Whilst running memory performance benchmarks, `FFT` and `NAS Parallel Benchmarks (NPB)` on CNK, Linux using 4 KiB pages, Linux using 64 KiB pages and Linux using Big Memory, they show that the performance of their implementation was very similar to that of CNK. Whilst performing random memory access patterns, CNK would achieve 44.70 MiB/s, whilst Linux with Big Memory achieved 44.68 MiB/s. When using 4 KiB and 64 KiB, the performance achieved was 14.39 MiB/s and 16.40 MiB/s, respectively. For FFT, Linux with Big Memory suffered a 0.008% performance loss compared to CNK. Sub 1% performance variations have been observed for NPB. Even when scaling up experiments, they observed a slowdown of well under 1%, thus showing the feasibility of Linux using very large pages for HPC compute nodes.

2.6 Evaluation

In this section we provide the evaluation of Transparent HugePages on ARM architecture-based systems, starting with the hardware and software setup, methodology, applications and parameters including a summary of results.

2.6.1 Hardware setup

We ran our experiments on Arndale boards [29], with the following configurations:

- Arndale board
  - Version 2.0 (Samsung Exynos 5 Dual [30], 1.7 GHz dual-ARM Cortex-A15, 32 KB 2-way set associative L1 data cache and 32 KB 2-way set associative L1 instruction cache, and a 1 MB 16-way set associative L2 cache)
  - 2 GB LPDDR3 800 MHz RAM
  - rootfs, including home directories, served over NFS
  - connection to server via 100 Mbps Ethernet, through shared switch
  - Linaro Linux kernel 3.7 [31] using a Linaro Ubuntu Server 13.01 userspace [32]
  - Due to overheating issues, we capped the frequency at 1.2 GHz. For all the results presented here, the board did not have a heatsink, was running with the performance governor, and had no throttling.

We chose a recent kernel that has support for HugeTLB and THP. We used the latest version of working kernel and userspace available at the time of running the experiments.

Limitations One of Arndale board's limitations is overheating. Running any application on one core alone can drive the temperature above the critical threshold. This happens on the stock board. A solution is to add a passive heatsink. Another one is to enable throttling in order to keep the board at about 85 °C. Without either, the board can easily reach temperatures of 90–100 °C.
Another limitation of the setup is the fact that the board only has 2 GB RAM. Given its 512-entry L2 TLB, and 2 MB hugepages, it can hold in TLB references that cover half its memory. Therefore it would be more interesting to see how performance is affected when the system has more RAM.

2.6.2 Software Setup

Sample HPC applications Benchmarks give a good indication of each optimization's impact, due to their specialized nature. However, this is not a guarantee that real HPC applications are going to obtain similar performance improvements, or whether the optimizations are even relevant. As a result, we explored the work in the context of three HPC mini-applications: CoMD, HPCG and Lulesh.

CoMD [1] is a reference implementation of typical classical molecular dynamics algorithms and workloads. It is created and maintained by ExMatEx: Exascale Co-Design Center for Materials in Extreme Environments. The code is intended to serve as a vehicle for co-design by allowing others to extend and/or reimplement it as needed to test performance of new architectures, programming models, etc.

HPCG [2] is a software package that performs a fixed number of symmetric Gauss-Seidel preconditioned conjugate gradient iterations using double precision (64 bit) floating point values. Integer arrays have global and local scope (global indices are unique across the entire distributed memory system, local indices are unique within a memory image).

LULESH [33] is a highly simplified hydrodynamics application, hard-coded to only solve a simple Sedov blast problem with analytic answers – but represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications.

A list of what parameters were used for each application can be found in this section. When using large pages, all applications can be fully covered by the L2 TLB on an ARM Cortex-A15.

CoMD, Lulesh, HPCG parameter tables:

CoMD	
Problem size	20 20 20 40 40 40
Potential	LJ EAM
Grid	1 x 1 x1 1 x 2 x 1
Total memory usage LJ	17-96 MiB
Total memory usage EAM	30-210 MiB

HPCG	
Problem size	104 104 104
Runtime	50
Total memory usage	786 MiB
Lulesh	
Problem size	30
MPI tasks	1
Num threads	2
Total memory usage	15.4 MiB

2.6.3 Compilers

Unless otherwise specified, all applications were built with a recent, if not the latest, version of a stable compiler available at the time of running the experiments. For the Arndale board it is a Linaro gcc-4.7.2.

2.6.4 Compilation flags

In this section we provide the compilation flags we used to build all the mini-applications. We chose these ones, as we aimed to enable NEON and auto-vectorization in all cases, and other optimizations guided by what was suggested for various applications.

2.6.5 Arndale board

All applications were built with the following compilation flags:

```
        -O3 -fomit-frame-pointer -funroll-loops -mcpu=cortex-a15
-mtune=cortex-a15 -mfloat-abi=hard -mfpu=neon -funsafe-math-optimizations
            -ftree-vectorize -march=armv7-a -DL2SIZE=1048576
```

2.6.6 Processor affinity

For the purpose of these results, we have not set the processor affinity for any benchmark, therefore it defaults to whether the application sets it or not.

2.6.7 MPI library

We used the MPICH [34] implementation of MPI version mpich2-1.5.

2.6.8 Custom ARM architecture optimizations

For the purpose of these tests, none of the applications have been customized for the ARM architecture, other than using the compilation flags described in Section 2.6.4. No source code modifications have been made and the applications are used as they are out of the box.

2.6.9 Methodology

For benchmarking the system, we used the following method:

1. Build a Linux kernel with Transparent HugePage support by enabling `CONFIG_TRANSPARENT_HUGEPAGE` and `CONFIG_TRANSPARENT_HUGEPAGE_ALWAYS`.

2. Boot the baseline system with no hugepage support, by running once it has booted: `echo "never" > /sys/kernel/mm/transparent_hugepage/enabled`.

3. Run the applications by using the custom-made scripts, with well-documented run parameters.

4. Enable Transparent HugePage support by running: `echo "always" > /sys/kernel/mm/transparent_hugepage/enabled`.

5. Re-run the applications as before, by using the same scripts and parameters.

2.6.10 Results for HPC mini-applications on ARM

CoMD

Board	Potential	nx	ny	nz	nSteps	Baseline Atom Update Rate (us/atom/task)	THP Atom Update Rate (us/atom/task)
Arndale board	LJ	20	20	20	100	7.52	7.52
		40	40	40		7.23	7.36
	EAM	20	20	20		15.01	15.02
		40	40	40		13.97	14.06

Table 9: CoMD single-threaded performance results

Single-threaded version

MPI version This version is making use of both ARM Cortex-A15 cores.

Board	Potential	nx	ny	nz	nSteps	Baseline Atom Update Rate (us/atom/task)	THP Atom Update Rate (us/atom/task)
Arndale board	LJ	20	20	20	100	7.98	7.99
		40	40	40		7.54	7.61
	EAM	20	20	20		15.77	15.82
		40	40	40		14.48	14.49

Table 10: CoMD multi-threaded performance results

In neither single-threaded, nor multi-threaded case does CoMD show any significant performance variability. This is shown in Tables 9 and 10.

Board	nx	ny	nz	Baseline Exec time (s)	Baseline GFLOP/s	THP Exec time (s)	THP GFLOP/s
Arndale board	104	104	104	77.7	0.264	77.3	0.266

Table 11: HPCG performance results

HPCG HPCG shows negligible performance variability between the baseline and hugepage cases. This can be seen in Table 11.

Problem Size	MPI tasks	Num processors	Iteration Count	Final Origin Energy
30	1	2	932	2.025075e+05

Table 12: Lulesh common parameters

	Baseline		THP	
Board	Elapsed time (s)	FOM (z/s)	Elapsed time (s)	FOM (z/s)
Arndale board	180	139	180	139

Table 13: Lulesh performance results

Lulesh As seen in Table 13, there is no variation between running Lulesh with small and large pages.

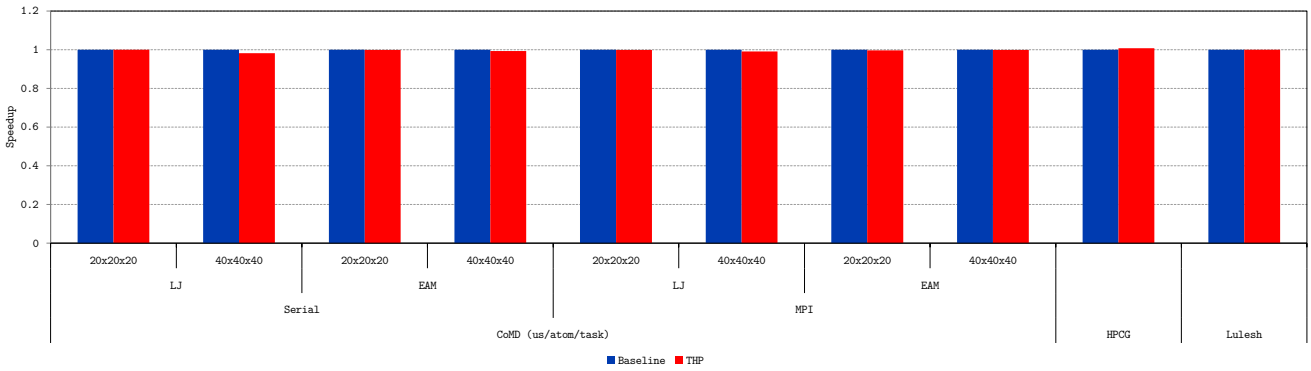

Figure 3

We did not observe any significant performance variation between using small and large pages when running the HPC mini-applications. This is shown in Figure 3.

2.7 Comparison with contemporary architectures

In this subsection we provide an overview comparison with contemporary architectures, by running the same experiments that we did on ARM architecture-based boards, on an x86_64 system.

2.7.1 Hardware setup

All experiments were run on an Intel Core i7-3770 (4 cores, 8 threads, 32 KB 8-way set associative L1 data and, respectively, instruction cache, 256 KB 8-way set associative L2 cache, 8 MB 16-way set associative L3 cache) running at 3.4 GHz. The system had 32 GB DDR3 1600 MHz RAM, and was running Ubuntu 12.04 LTS, kernel v3.2.0-48 x86_64.

2.7.2 Software setup

The applications, parameters and methodology, used for comparing with contemporary architectures were the same as for running on the ARM architecture. The only difference was in compilation flags.

2.7.3 Compiler

The compiler we used for all x86_64 applications and mini-applications is a gcc-4.6.3.

2.7.4 Compilation flags

We tried matching the compilation flags as closely as possible between x86_64 and ARMv7-A. We used the following flags for all applications:

```
-O3 -fomit-frame-pointer -funroll-loops -funsafe-math-optimizations
            -ftree-vectorize -mtune=corei7-avx -mavx
```

2.7.5 Other libraries

For the purpose of these tests we used the default `libatlas` library found on an Ubuntu 12.08 Linux distribution and a custom-built mpich2-1.5 MPI library.

2.7.6 Results for HPC mini-applications on x86_64

CoMD

Board	Potential	nx	ny	nz	nSteps	Baseline Atom Update Rate (us/atom/task)	THP Atom Update Rate (us/atom/task)
x86_64	LJ	20	20	20	100	1.816	1.806
		40	40	40		1.723	1.720
	EAM	20	20	20		2.906	2.863
		40	40	40		2.706	2.686

Table 14: CoMD single-threaded performance results

Single-threaded version

MPI version

This version is making use of 2 hardware threads.

Board	Potential	nx	ny	nz	nSteps	Baseline Atom Update Rate (us/atom/task)	THP Atom Update Rate (us/atom/task)
x86_64	LJ	20	20	20	100	1.906	1.899
		40	40	40		1.796	1.796
	EAM	20	20	20		3.093	3.030
		40	40	40		2.800	2.803

Table 15: CoMD multi-threaded performance results

When running CoMD in both single-threaded and multi-threaded versions, as seen in Tables 14 and 15, respectively, we observed no significant change in performance between running with 4 KiB and 2 MiB pages.

HPCG

				Baseline		THP	
Board	nx	ny	nz	Exec time (s)	GFLOP/s	Exec time (s)	GFLOP/s
x86_64	104	104	104	40.75	1.516	40.59	1.522

Table 16: HPCG performance results

As seen in Table 16, there is no significant performance variability between the baseline and hugepage versions.

Lulesh

Problem Size	MPI tasks	Num processors	Iteration Count	Final Origin Energy
30	1	1	932	2.025075e+05

Table 17: Lulesh common parameters

	Baseline		THP	
Board	Elapsed time (s)	FOM (z/s)	Elapsed time (s)	FOM (z/s)
x86_64	23.4	1075.080	23.6	1064.172

Table 18: Lulesh performance results

As seen in Table 18, there is no significant performance variability between the 4 KiB and 2 MiB page versions.

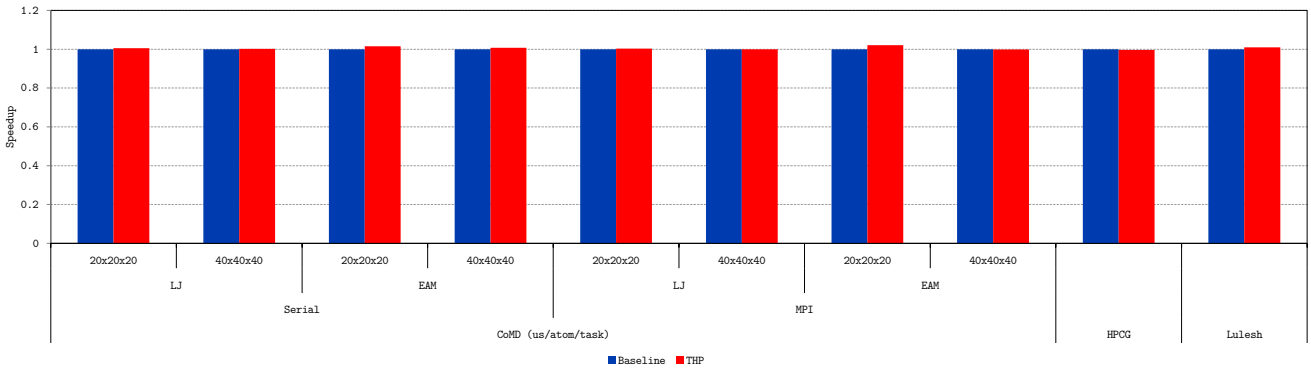

Figure 4

HPC mini-applications None of the HPC mini-applications shown in Figure 4 show any significant performance variation. Any couple of percent variation is attributed to noise.

2.8 Observations

Overall, we did not observe any major differences in the relative performance between using small and large pages on x86_64 or ARMv7-A.

Using large pages is expected to improve performance by increasing the amount of memory covered by the TLB and decreasing the size of the page tables. This should result in fewer page faults and fewer cache misses due to page table accesses.
Thus the memory access pattern for a particular application will determine whether using large pages produces an improvement in performance. Applications with good data locality will not produce any significant performance improvement when using large pages. However, when there is little data reuse, i.e sparse or random memory accesses, then we would expect using large pages to make a significant difference.
We believe that good data locality and relatively small memory usage is the reason that we saw no change in performance when using large pages on either the ARMv7-A or x86_64 based systems for the applications.

2.9 Conclusions

In this report we presented an evaluation of Transparent HugePages running on the ARM architecture, primarily from an HPC perspective.
For reference, we offered results running on a contemporary platform, more specifically an Intel x86_64 desktop machine using the same HPC mini-applications.
Overall, we believe hugepages are beneficial to use on ARMv7-A. We expect to obtain better cache utilization, memory latency and overall performance from using hugepages. We have seen this reflected in the results previously reported in MontBlanc but for more "real" applications the effects are not yet proven.

2.10 Further work

2.10.1 AArch64 evaluation

As most HPC and server applications are 64-bit, we aim at producing a similar evaluation for AArch64. This will enable us to look at more page sizes. We will also take this opportunity to look at other features such as virtualization and how it is impacted by Transparent HugePages.

2.10.2 Testing on systems with more RAM

A constraint of the setup is the fact that the Arndale board only has 2 GiB RAM. Given its 512-entry L2 TLB, and 2 MiB hugepages, we can hold in TLB references that cover half its memory region. The Mont-Blanc prototype system has 4 GiB RAM per SoC so it would be interesting to see how performance is affected when the system has more RAM.

2.10.3 Larger memory usage

As well as testing on a system with more RAM, we want to run applications that attempt to use much more of the system RAM. We will increase the memory usage of the HPC mini-applications (by increasing the problem size) to try and show the effects of using larger pages on performance.

3 OmpSs@cluster runtime system

We have ported OmpSs [35], and specifically its cluster flavor [36, 37] to the ARMv7-A architecture. In this section, we show the steps taken and a preliminary evaluation with the matrix multiply benchmark.

3.1 Development

Initially, the OmpSs@cluster runtime system (Nanos++ [38]) was already working on the Intel architecture. Deliverable D4.2 [39] shows the latest development that has been done in OmpSs@cluster regarding the caching system and affinity scheduler. These features are already incorporated in the ARM version.
In order to perform the porting, we followed these steps:

- GASnet [40, 41] has been adapted to be used for the ARM architecture. We have used the odroid system to test GASnet on ARM, and solve a few issues that we observed:
  - GASNet running on top of the UDP protocol
    - was binding itself to the 0.0.0.0 IP address, and this was not working on the odroid system.
      - Proper binding to the current node IP address solved this issue.
    - reported errors regarding network congestion, and it was unable to deal with them.
  - GASNet running on top of the MPI conduits (MPICH2, and OpenMPI) reported errors regarding truncated messages.
    - This problem was solved by reducing the maximum amount of data that Nanos++ sends at once to the network. It was set to 2 Gbytes of data for Infiniband in the Intel architecture, and it has been reduced to 32 Kbytes for the Ethernet on ARM.
- Nanos++ has been compiled using GASNet
  - We have selected to use the version of GASNet running on OpenMPI.
  - It allowed us to run matrix multiplication on 1 to 8 nodes, and matrix sizes of 2048x2048 elements.

3.2 Evaluation

The preliminary evaluation of the porting of OmpSs@cluster to the ARMv7-A architecture has been done in the Odroid platform, with these caracteristics:

- Up to 8 nodes, Odroid board, SoC Samsung Exynos 5 Octa 5410
- CPU Cortex A15@1.6Ghz quad core
- 2 Gbytes RAM

We took advantage of having the benchmarks explained in Deliverable D4.2 [39] running on the Intel architecture, to get matrix multiplication easily running on the ARM platform.
Figure 5 shows the execution time obtained in the blocked matrix multiply benchmark with OmpSs@cluster in this platform. We have executed two versions of the benchmark:

- Single level, in which parallelism is exploited by spawning a single level of tasks, as it is usually the case in shared memory environments.
- Multi level, in which a first level of tasks is spawned for each block of the result (C), and inside a second level of parallelism is spawned to compute on the different blocks of A and B.

Both versions have been executed with the default breadth-first scheduling policy and also with the cluster-specific affinity policy.
Results show that the affinity scheduling policy provides benefits due to the increase of data locality. In addition, in this environment, it is not clear that the multi-level solution gets benefits over the single level. It is part of our future work to investigate the reason of this issue.

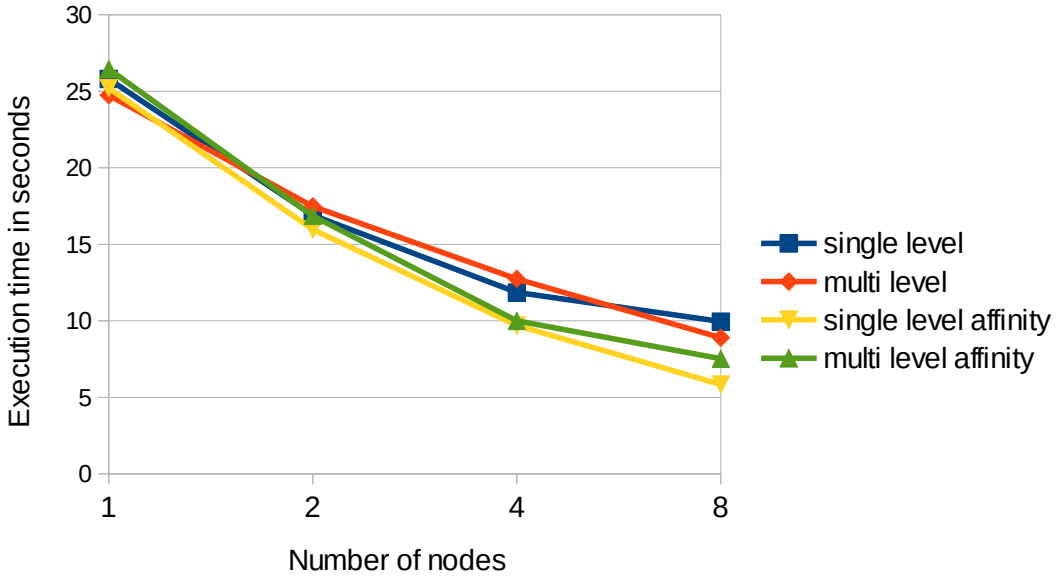

Figure 5: Execution time of the matrix multiply benchmark with OmpSs@cluster in the ARMv7-A architecture

3.3 Conclusions

We have shown that the OmpSs@cluster version has been ported to the ARMv7-A architecture and it is successfully running. We have shown the preliminary evaluation of matrix multiply in the Odroid platform.
It is part of our future work to determine the issues detected on the porting and evaluation. It is necessary to know if we can increase the message size limits on MPI, determine why the multi-level version of matrix multiply is not performing better than the single-level version, and also perform the comparison with the corresponding MPI version of matrix multiply, and the rest of the benchmarks.

References

[1] "ExMatEx, CoMD Proxy Application." `http://www.exmatex.org/comd.html`. accessed 05 Sept 2014.

[2] Heroux, M.A., Dongarra, J., Luszczek, P., "*HPCG Technical Specification*, Sandia Report SAND2013-8752 (2013)." `https://software.sandia.gov/hpcg/doc/HPCG-Specification.pdf`.

[3] "ARM, Streamline." `http://ds.arm.com/ds-5/optimize/`. accessed 28 Sept 2014.

[4] "GDB: The GNU Project debugger." `http://www.gnu.org/software/gdb/`. accessed on 28 Sept 2014.

[5] ARM Ltd., NEON Programmer's Guide. 2013. sec 5.2.6.

[6] Yao, Z., Wang, J.S, Liu, G.R., Cheng, M., "*Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method*," in *Computer Physics Communications volume 161, Issues 1-2*, 2004.

[7] Plimpton, S., "*Fast Parallel Algorithms for Short-Range Molecular Dynamics*," in *Journal of Computaional Physics 117*, 1995.

[8] Grest, G.S., Dunweg, B., Kremer, K., "*Vectorized link cell Fortran code for molecular dynamics simulations for a large number of particles*," 1989.

[9] Verlet, L, "*Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules*," in *Phys. Rev. 159, 98*, 1967.

[10] Allen, M.P., Tildesley, D.J., Computer Simulation of Liquids. Oxford Univ. Press, New York, 1990.

[11] Zhang, X., Yang, C., Liu, F., Liu, Y., Lu, Y., "*Optimizing and Scaling HPCG on Tianhe-2: Early Experience*," in *Algorithms and Architectures for Parallel Processing*, 2014.

[12] Kumahata, K., Minami, K., Maruyama, N., "*HPCG on the K computer*," 2014. ASCR HPCG Workshop.

[13] Preclik, T., Bartuschat, D., "*Gauss Seidel Parallelization*, University Erlangen-Nuremberg (2009)." `www10.informatik.uni-erlangen.de/en/Teaching/Courses/WS2009/SiWiR/exerciseSheets/ex02/rbgs.pdf`. accessed 08 Sept 2014.

[14] "D5.4 Report on Tuned Linux-ARM kernel and Delivery of KErnel Patches to the Linux Kernel." urlhttp://montblanc-project.eu/sites/default/files/D5.4Report_on_Tuned_Linux-ARM_kernel...v1.0.pdf.

[15] "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition." `http://arminfo.emea.arm.com/help/topic/com.arm.doc.ddi0406c.c/DDI0406C_C_arm_architecture_reference_manual.pdf`. Accessed 19 June 2014.

[16] "HugeTLB initial kernel support." `http://linuxgazette.net/155/krishnakumar.html`. Accessed 19 June 2014.

[17] "Transparent HugePages initial kernel support." `http://lwn.net/Articles/423584/`. Accessed 19 June 2014.

[18] "7-cpu ARM Cortex-A15." http://www.7-cpu.com/cpu/Cortex-A15.html. Accessed 19 June 2014.

[19] "LMbench - Tools for Performance Analysis." http://www.bitmover.com/lmbench/. Accessed 19 June 2014.

[20] K. Milfeld, K. Goto, A. Purkayastha, C. Guiang, and K. Schulz, "Effective use of multi-core commodity systems in hpc," *way*, vol. 2, no. 2, p. 2, 2007.

[21] "libhugetlbfs library." http://libhugetlbfs.sourceforge.net/. Accessed 19 June 2014.

[22] C. Marinas, "ARM: mm: Transparent huge page support for LPAE systems.." https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=8d962507007357d6fbbcbdd1647faa389a9aed6d. Accessed 19 June 2014.

[23] Catalin Marinas, "ARM: mm: HugeTLB support for LPAE systems.." https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=1355e2a6eb88f04d76125c057dc5fca64d4b6a9e. Accessed 19 June 2014.

[24] R. Noronha, "Improving scalability of openmp applications on multi-core systems using large page support," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8, 2007.

[25] C. McCurdy, A. Cox, and J. Vetter, "Investigating the tlb behavior of high-end scientific applications on commodity microprocessors," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pp. 95–104, April.

[26] ARM, "ARM Cortex-A15 MPCore technical reference manual." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/index.html. Accessed 19 June 2014.

[27] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber, "Evaluating the effect of replacing cnk with linux on the compute-nodes of blue gene/l," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 165–174, ACM, 2008.

[28] K. Yoshii, K. Iskra, H. Naik, P. Beckmanm, and P. C. Broekema, "Characterizing the performance of "big memory” on blue gene linux," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, (Washington, DC, USA), pp. 65–72, IEEE Computer Society, 2009.

[29] InSignal, "Arndale 5 Base Board System Reference Manual." http://www.arndaleboard.org/wiki/downloads/supports/BaseBoard_Specification_Arndale_Ver1_0.pdf. Accessed 19 June 2014.

[30] "Samsung Exynos 5 Dual Product page." http://www.samsung.com/global/business/semiconductor/minisite/Exynos/products5dual.html. Accessed 19 June 2014.

[31] Linaro, "Git repository for Linaro kernel for Arndale." gitclonegit://git.linaro.org/people/ronynandy/u-boot-arndale.git. Accessed 19 June 2014.

[32] Linaro, "Linaro Ubuntu Server usespace 13.01 for Arndale." `https://releases.linaro.org/13.01/ubuntu/arndale/linaro-quantal-server-20130130-258.tar.gz`. Accessed 19 June 2014.

[33] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[34] "MPICH home page." `http://www.mpich.org/`. Accessed 19 June 2014.

[35] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a Proposal for Programming Heterogeneous Multi-core Architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[36] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Proceedings of the 17th International Conference on Parallel and Distributed Computing*, Euro-Par '11, pp. 555–566, Springer Berlin Heidelberg, 2011.

[37] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayuguade, and J. Labarta, "Productive programming of gpu clusters with ompss," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, (10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314, USA), pp. 557–568, IEEE Computer Society, 2012.

[38] "Nanos++ Programming Model." `http://pm.bsc.es/nanox`.

[39] A. Miranda, R. Nou, T. Cortes, J. Bueno, and X. Martorell, "Preliminary report on OmpSs Extensions and Storage Tuning," *Mont-Blanc Deliberable D4.2*, 2014.

[40] D. Bonachea, "Gasnet specification v1.1," Tech. Rep. CSD-02-1207, 2002.

[41] "GASNet website." `http://gasnet.lbl.gov/`. Accessed 12 September 2014.