Performance Measurement on Heterogeneous Processors with PAPI

Willow E. Cunningham
Electrical and Computer Engineering
University of Maine
Orono, Maine, USA

Vincent M. Weaver
Electrical and Computer Engineering
University of Maine
Orono, Maine, USA
vincent.weaver@maine.edu

***Abstract*—**
Modern computer processors improve their computing power by having multiple cores. Traditionally these cores were homogeneous: many identical cores with the same capabilities. Instead it is possible to create processors that have heterogeneous (or hybrid) cores, where the various cores have differing capabilities. This can lead to energy savings and other efficiencies, but complicates performance analysis. Heterogeneous cores have been common for years in embedded ARM processors; recently support has appeared in x86 desktop processors as well. It is likely that before long server and high-performance systems will also gain hybrid cores.

We look at current Linux support for heterogeneous processors and detail the various problems encountered when adding support for them to the PAPI performance measurement library.

***Index Terms*—Linux perf, perf_event, PAPI, heterogeneous processors, hybrid cores, performance measurement, big.LITTLE**

I. Introduction

The processors inside of modern computers gain performance by having multiple computing cores. Properly written software can increase performance by running code across multiple cores at once. Traditionally these multi-core systems use identical (or homogeneous) cores. This greatly simplifies the design of the system and also makes writing code simpler.

It is possible to instead design a hybrid (or heterogeneous) system where a variety of different core types work together while providing differing abilities. Typically this is done for power-saving reasons: you can have fast (but power-hungry) cores used when performance is needed, but smaller more power-efficient cores used when on battery power or when running less critical applications.

Note that we are specifically describing heterogeneous CPU cores; there is another type of heterogeneous computing where CPU cores are paired with GPU (graphics) cores. This is a related topic but not something covered in this work.

A. Availability of Heterogeneous CPUs

Heterogeneous systems became widely used with the introduction of ARM's big.LITTLE technology in 2011 [1], [2]. This type of CPU was primarily found in embedded devices, phones, tablets and laptops. More recently support has come to x86 desktops and laptops. Intel has introduced systems that have what they call P-core (performance) and E-core (efficiency) processors starting with their 12th generation Alder Lake systems [3]. Intel has yet to announce a server chip with heterogeneous cores, but their upcoming Sierra Forrest and Granite Rapids chips will exclusively be E or P core respectively, showing that there is a demand for differing core capabilities in modern server systems. AMD has not announced plans for heterogeneous processors yet, but they have contributed patches to the Linux kernel that appear to be preparing for just such a release [4].

B. Challenges with Heterogeneous CPUs

There are many challenges that come from heterogeneous systems, with the most prominent being job scheduling. Multicore thread scheduling on homogeneous systems is already a difficult task, the situation has only gotten worse with the introduction of heterogeneous processors.

The scheduler must be aware of the differing capabilities of the cores and try to make intelligent choices of how best to allocate resources while trying to maintain optimal power and performance usage. Overhead in scheduling is a key bottleneck in operating system performance so these decisions must be made as quickly as possible and without much code being run (currently Linux uses the "Completely Fair Scheduler" which completes in $O(logN)$ time). The low level details of heterogeneous processors are beyond the scope of this work, but details can be found in other works [5]–[9]. Often these heterogeneous-aware schedulers make use of hardware performance counters to track how all the cores are behaving.

C. Performance Measurement of Heterogeneous CPUs

Most modern CPUs provide access to hardware performance counters that allow collecting detailed performance information that can be used for code optimization. Often hundreds to thousands of possible events can be measured, covering all aspects of the underlying computer architecture, including critical metrics like total cycles, retired instructions, branch misses, and cache misses.

Unlike some CPU interfaces that are standardized and remain the same from one processor generation to another (or even across vendors) the performance counter interface tends to vary from CPU to CPU. Which events are available can

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00195

TABLE I
HARDWARE CONFIGURATION OF THE RAPTOR LAKE SYSTEM

CPU	13th Gen Intel(R) Core(TM) i7-13700
P-cores (performance)	8 (16 threads) @2.10-5.10 GHz
E-cores (efficiency)	8 @1.50-4.10 GHz
Memory	32GB DDR5, 4.4G T/s

TABLE II
BENCHMARK PERFORMANCE COMPARISON

Enabled cores	OpenBLAS HPL	Intel HPL	% Change
E only	188.62 Gflops	198.95 Gflops	+5.4%
P only	356.28 Gflops	392.89 Gflops	+10.3%
P and E	290.51 Gflops	457.38 Gflops	+57.4%

vary not only by architecture, but also by vendor and even by CPU model. This was already a challenge when doing cross-platform performance measurements, but it becomes even more complicated on heterogeneous systems. The various core types can be of different microarchitectures and thus the cores can have different event interfaces and availability. (An example of this, Intel top-down events are only available on the Raptor Lake's P-cores but not the E-Cores). Most existing tools assume a system will only have a single, core, set of CPU events and may fail when encountering a heterogeneous setup. Tools that are aware of varying cores must handle measuring processes that might switch core types mid-run and report the results back to the user in an understandable way.

D. PAPI and Heterogeneous CPUs

The Performance API (PAPI) is a cross-platform library widely used to measure performance data in high performance computing [10]. When used on Linux systems it uses the underlying perf_event subsystem to handle access to the counters provided by the CPUs [11].

The current PAPI 7.1 release does not support heterogeneous processors. In this work we investigate how the Linux kernel handles heterogeneous systems, and then describe the changes being made to PAPI to enable support for these cores. By adding the support to PAPI any external tools that build on it should gain heterogeneous support. While we focus on the support needed for the PAPI library, the work described should be useful for other low-level performance tools that need to add heterogeneous CPU support.

II. MOTIVATION

We investigate the performance of benchmarks on heterogeneous CPUs to show why it is advantageous to have hybrid CPU-aware performance tooling.

A. Comparing HPL Varieties on Intel Raptor Lake

Software designed with homogeneous processors in mind is not necessarily well optimized for systems with heterogeneous processors. We compare two versions of the High Performance Linpack (HPL) benchmark: HPL [12] compiled from source using OpenBLAS (OpenBLAS HPL) [13], and Intel Optimized LINPACK (Intel HPL) from the Intel oneAPI Math Kernel Library (MKL) [14].

1) Experimental Setup: We compare the two HPL benchmarks on a desktop PC equipped with a heterogeneous Intel Raptor Lake CPU as described in Table I. The operating system running on the Raptor Lake machine is Debian GNU/Linux 12 (bookworm), using Linux 6.7.12+bpo-amd64 kernel. The gcc compiler version 12.2.0 is used to compile the code. While the 8 P-cores each have two hardware threads, all HPL runs are limited to one thread per core.

OpenBLAS HPL is built using OpenMPI v4.1.4 and OpenBLAS version 0.3.27.dev, compiled from source for the best performance. Intel HPL is built using the distributions of MPI 2021.12 and MKL 2024.2 in oneAPI.

2) Configuring HPL: Proper tuning of HPL via editing the `HPL.dat` file can have a significant impact on its results. The most important parameters for a single node computer are the problem size N and block size NB. Since the Raptor Lake machine has only one node, the process grid shape parameters P and Q are both necessarily set to 1. The HPL documentation recommends that N be selected to use 80% of the system's memory, although in practice this can vary.

In order to select a good N and NB value pair OpenBLAS HPL is run 16 times using N values calculated with the β-approach presented by Krpić, Loina, and Galba [15] for 70%, 75%, 80%, and 85% memory usage paired with NB values of 64, 128, 192, and 256. From these runs it is found that HPL performs best on the Raptor Lake system with $N = 57024$ and $NB = 192$. These values are used in the `HPL.dat` file for both of the tested HPL benchmarks.

3) Data Collection: A python script starts the HPL run and then polls CPU core frequency, thermal zone temperatures, and Running Average Power Limit (RAPL) [16] energy counter values at a rate of 1 Hz until the run is completed. The script performs multiple identical HPL runs, waiting for the CPU package temperature to settle at 35 °C before each run to ensure that the effects of thermal throttling are similar across each run. The Linux `perf` command is used to start HPL each time, and in doing so some performance counter data is also collected. Another script is then used to aggregate the data into an averaged run for analysis.

For both OpenBLAS HPL and Intel HPL data is collected as described previously and averaged over 10 runs. For both of the benchmarks `HPL.dat` is configured with $N = 57024$, $NB = 192$, $P = 1$, and $Q = 1$.

4) Raptor Lake HPL Results: A comparison of the results of the two benchmarks is shown in Table II. Intel's Optimized HPL outperformed OpenBLAS HPL for each set of cores, as can be expected of any software running on the specific architecture it has been designed for. Of particular note is the 57.4% increase in performance between OpenBLAS HPL and Intel HPL for the all-core runs.

OpenBLAS HPL performed 18.5% worse when run on all cores than when limited to only the P-cores. This demonstrates that software that is not designed with heterogeneous processors in mind (like OpenBLAS HPL) can actually have their

TABLE III
HARDWARE COUNTER MEASUREMENTS FOR ALL-CORE RUNS

	OpenBLAS HPL		Intel HPL		% Change	
Core type	P	E	P	E	P	E
LLC missrate	86%	0.05%	64%	0.03%	-26.3%	-39.8%
% of total instructions	80%	20%	68%	32%	-	-

performance negatively affected by the presence of E-cores. Conversely, Intel HPL performed 16.4% better when given access to all cores than when it was limited to only the P-cores. This shows that with proper knowledge of low-level CPU behavior it is possible and beneficial to leverage all cores of a heterogeneous processor for high performance tasks.

For a closer look at the different behaviors between the two benchmarks when run on all cores the average measured core frequencies for both benchmarks are shown in Figure 1.

The high level of noise in core frequency seen in Figure 1(a) is also present for all configurations of OpenBLAS HPL, and is therefore not likely the root cause of the significant performance increase obtained by Intel HPL for its all-core runs. The two benchmarks differed in core frequencies for the all-core runs in that the median frequency of the P-cores was lower for Intel HPL (2.61 kHz) than for OpenBLAS HPL (2.94 kHz), and the median frequency of the E-cores was slightly higher for Intel HPL (2.32 kHz) than for OpenBLAS HPL (2.26 kHz). In summary, the heterogeneous core frequencies for Intel HPL were less dissimilar than for OpenBLAS HPL.

The initial spike in core frequencies seen in both Figures 1(a) and 1(b) starting at around $t = 110s$ and $t = 1s$ respectively correspond to the RAPL short term power cap (219 W) time limit being exceeded. For both benchmarks, the majority of the run is capped by the long term power limit of 65 W, as shown in Figure 2.

OpenBLAS HPL is unable to reach the full short term power cap, peaking at 165.7 W before quickly dropping to the long term limit for the rest of the run. Neither benchmark is affected by thermal throttling as the power limits and adequate cooling prevent the cores from reaching their maximum allowed package temperature of 100 °C [17].

Values derived from performance counter measurements collected with perf for both benchmarks are shown in Table III, including the miss rate of the last level cache (LLC) for each core type and the percentage of instructions for each benchmark that were run by each core type.

It can be seen that Intel HPL ran more of its code on the P-cores when compared to OpenBLAS HPL. Additionally, the LLC missrate for both core types were greatly reduced for Intel HPL, particularly that of the E-cores. As discussed in Stepanovic et al. [18] it is usually optimal to relegate jobs with a high LLC miss rate to the E-cores. Intel HPL takes advantage of this fact to achieve greatly improved performance.

TABLE IV
HARDWARE CONFIGURATION OF THE ORANGEPI 800 SYSTEM

CPU	Rockchip RK3399 SoC
big cores	2 ARM Cortex-A72 @1.8 GHz
little cores	4 ARM Cortex-A53 @1.4 GHz
Memory	4GB LPDDR4

B. ARM big.LITTLE Exploration

In addition to the Raptor Lake work, we have investigated behavior on an ARM64 heterogeneous platform.

1) Experimental Setup: We run experiments on an Orange Pi 800 system, which has six cores: two Cortex-A72 "big" cores, and four Cortex-A53 "LITTLE" cores as described in Table IV. We run HPL on this system, compiled against OpenBLAS with gcc 11.4. The system is running the OrangePi Ubuntu Jammy distribution with a 64-bit Linux 5.18.5-rk3399 kernel.

2) ARM big.LITTLE Results: As seen in Figure 3 the big cores quickly ramp up to maximum frequency, but not for long, as the temperature quickly gets too high and the cores are scaled back down. Figure 4 shows that the big cores throttle so quickly that using four little cores can give better overall performance than using just the big cores, and using all six cores gives only a slight improvement over leaving the big cores unused. This behavior is more complex than one would usually see while running an embarrassingly parallel benchmark like HPL on homogeneous cores, and analysis would benefit greatly from performance tools that are heterogeneous processor-aware.

III. RELATED WORK

Heterogeneous processing is a mature technology, especially the big.LITTLE ARM variant. There has been previous work looking at using performance counters on these systems, most using a wider variety of benchmarks than we do. Unlike the other previous work, we are able to look at both Intel and ARM heterogeneous systems.

The work closest to ours is Gupta et al. [19] who use PAPI with LLVM and Clang to comprehensively instrument and characterize individual components of a series of benchmarks on an ARM big.LITTLE system. The characterized data is used offline to train a classifier that determines optimal core configurations and frequencies at runtime. They claim to use PAPI for these measurements, but it is unclear if they have added heterogeneous support themselves, as at that time PAPI did not have support for big.LITTLE systems.

Vasilakis et al. [20] look at the performance of single-threaded benchmarks on the big and little cores of an ARM big.LITTLE processor, using Instructions Per Cycle (IPC) derived from hardware counters as the primary performance indicator.

Whitehouse et al. [21] use performance counters to investigate the core utilization and the impacts of branch misprediction on heterogeneous multicore processors over an array of benchmarks and everyday mobile device use cases such as playing a YouTube video.

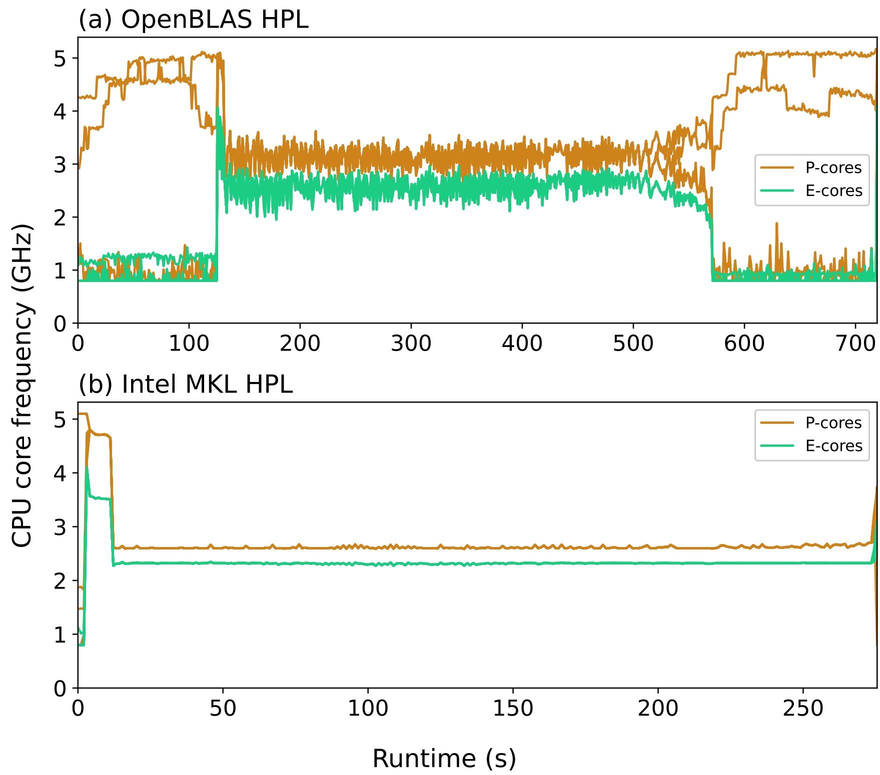

Fig. 1. Measured core frequencies on the Intel Raptor Lake system for both variants of HPL, run on all cores.

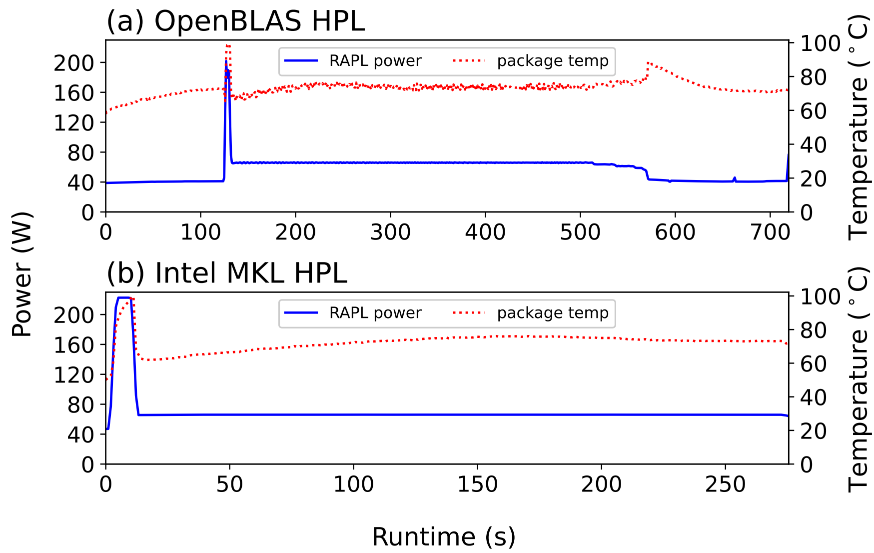

Fig. 2. Measured power and package temperature on the Intel Raptor Lake system for both variants of HPL, run on all cores.

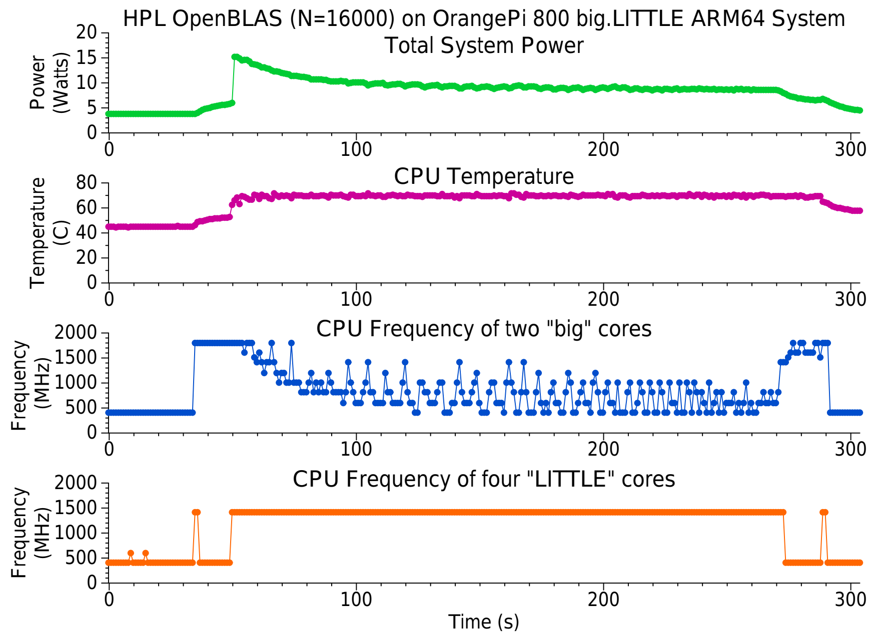

Fig. 3. Frequency scaling behavior on an ARM64 big.LITTLE system. HPL running on the big cores quickly leads to throttling due to temperature overhead. Most of the computation ends up happening on the LITTLE cores. Power is measured with a WattsUpPro Power meter.

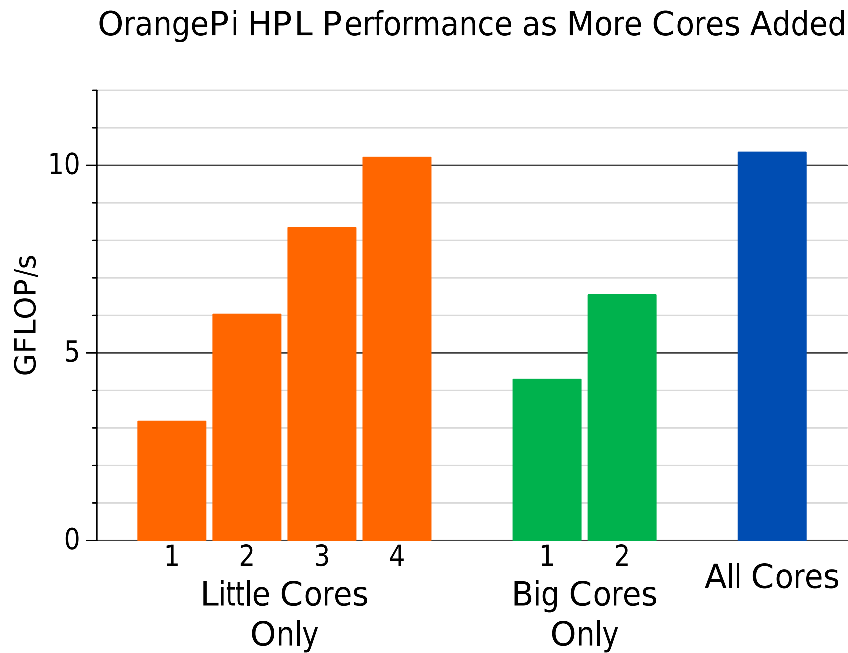

Fig. 4. Due to temperature related frequency throttling HPL running on all four little cores completes faster than running on both big cores. Running on all six cores provides a minimal improvement over just running on the four little cores.

Fernandez et al. [22] describe monitoring cache misses with performance counters to assess the suitability of heterogeneous multicore processors for real-time embedded applications. Stepanovic et al. [18] also look at cache misses, but with the goal of designing an efficient core selection model.

IV. Adding Heterogeneous CPU Support to PAPI

It might seem that adding heterogeneous support to the PAPI library would be a straightforward process. It turns out that full heterogeneous support will require a lot of intrusive changes to the codebase and redesigning some of the low-level interfaces.

A. Linux perf Heterogeneous Support

The Linux kernel has supported heterogeneous CPU support using perf_event since ARM big.LITTLE performance monitoring unit (PMU) support was added in the 4.10 kernel release (early 2017). Intel heterogeneous processor PMU support was merged with Linux 5.13 (2021), which has led to increased interest in having tools be more heterogeneous-aware.

The Linux perf_event interface handles heterogeneous CPUs by having a separate PMU type exported for each type of CPU core (usually there are two, but there exist ARM CPUs with three types and it is plausible even more will be supported someday).

The PMU type is passed into the kernel when using the `perf_event_open()` system call to open an event. A PMU's `type` value can be found in the `/sys/` directory: for example on Raptor Lake the E and P core PMUs (known to Intel as "atom" and "core" respectively) can be found in:

```
/sys/devices/cpu_atom/type
/sys/devices/cpu_core/type
```

When a regular performance event is opened, it is associated with a process thread and follows the thread wherever it is scheduled. By default the operating system scheduler is free to move the process to other cores, including ones of other types. The counter values are saved and restored at context switch to keep the counts consistent. This could be a problem on heterogeneous cores (an event might be measuring hardware features that do not exist on the new core, or the event itself might not exist at all there). To avoid this the kernel tracks the core type and only enables event counters if they match the core currently being run on. If you want to measure a common event like retired instructions across all possible cores that might run your code, you will need to open multiple events, one for each core type available on the system.

The Linux `perf` tool works in this way, by setting up multiple events on heterogeneous systems and reporting all of the results gathered. This is straightforward, but it does limit accuracy because even in the best case where the events being read from each core type share an event group, it will typically take at least two or more relatively high-latency read syscalls to gather all of the event values.

In theory adding heterogeneous support to PAPI is just a matter of allowing multiple PMU events per EventSet. However due to the underlying design of PAPI this requires some fairly intrusive changes to the perf_event component.

If the `perf` tool already supports heterogeneous events, why bother going through the trouble of fixing PAPI? PAPI provides many features not supported by `perf`, with the key advantage being that you can caliper your source code. This means you can go into your code and add `PAPI_start()` and `PAPI_stop()` calls around arbitrary chunks of code to do fine-grained measurement. The `perf` tool does not support this; currently it only supports gathering either aggregate (full-program) counts or else statistically sampled values.

B. PAPI support: Heterogeneous Core Detection

One major problem with heterogeneous processors on Linux is determining what types of cores are available and reporting this so the user or tools can take advantage of it. Currently Linux has no standard way of doing this. PAPI would like to have this information for a variety of internal reasons, but also so it can report it via the `PAPI_get_hardware_info()` interface, the device attribute interface, and via the sysdetect component.

For ARM big.LITTLE systems there are a few ways to attempt to get the info. There is:

```
/sys/devices/system/cpu/cpuX/cpu_capacity
```

which provides an opaque number between 0 and 1024. On machines with three types of cores often they are 250, 512, and 1024 but that is not guaranteed.

You can attempt to figure things out by looking at the CPU identification values in `/proc/cpuinfo` or `/sys/devices/system/cpu/cpuX/regs/identification`. This can help you differentiate if the cores are of different types (for example: Cortex-A53 vs Cortex-A72). This works on ARM, but Intel P/E-cores are not given separate family/model/stepping so this cannot be used generically.

Intel chips can report heterogeneous status via the `cpuid` instruction (leaf 1A bits EAX[31-24]) however this is Intel specific and not a general interface we can depend on for all types of cores.

The perf tool detects PMUs by searching the `/sys/devices/` directory, and inside any PMU subdirectory is a file called "cpus" that maps PMU to core. This has complications as well: on ARM the PMU names are set by the boot firmware and systems using devicetree (mostly embedded systems) and ACPI (mostly servers) can provide different names for the same PMU type.

Some tools resort to determining if heterogeneous cores are present by looking up the maximum possible frequency or cache sizes in:

```
/sys/devices/system/cpu/cpuN/cpufreq/
 cpuinfo_max_freq
/sys/devices/system/cpuN/cache
```

but this cannot always be guaranteed to work.

When initial support for Intel P/E-cores was added there was a patch proposed to try to create an official `/sys/devices/system/cpu/types` interface but it did not end up included [23]. There was recent discussion on the Linux-kernel mailing list that such a patch would be resubmitted for consideration.

C. PAPI support: libpfm4 Support

PAPI depends on the libpfm4 library [24] to provide event lists and to help create the perf_event attribute structure expected by the Linux kernel `perf_event_open()` syscall when specifying events.

Initially libpfm4 had no support for heterogeneous events, but support for Alder Lake and Raptor Lake P/E-cores was added after we requested it. The initial support had some bugs with the instructions retired E-core event, but we worked with the maintainer and got those fixed.

While libpfm4 now properly handles heterogeneous support on Intel P/E events it still does not work with ARM big.LITTLE systems due to the different way the ARM PMU support scans for PMUs. We have preliminary patches that allow for multiple PMUs on ARM, and we are working on getting a solution merged upstream. We also used a not-yet-merged patch to enable libpfm4 ARM Cortex-A72 support needed for our OrangePi system.

D. PAPI support: Multiple Default PMUs

The first thing needing to be fixed in PAPI was the possibility of having more than one default PMU. For backwards compatibility reasons PAPI has the notion of a "default" PMU that event names are searched in if no explicit PMU is specified. This was picked to be the primary CPU PMU, but on a heterogeneous system there are multiple CPU PMUs and PAPI did not handle this case well and would give an error or possibly even crash.

Support was added to handle a libpfm4 that reported multiple default cpus, such as the P and E-cores on Raptor Lake. We currently choose the "P" core as the default, but there is not a generic way of determining which of the core types should be default and for now it has to be hard-coded for each known heterogeneous CPU type.

E. PAPI support: Multiple PMUs in an EventSet

The most straightforward way to support heterogeneous systems is to allow a user to create an EventSet (PAPI's abstraction for a group of events that run at the same time) with equivalent events from both PMUs. For example on Raptor Lake to measure both retired instruction events, you would create an EventSet with these events:

```
adl_glc::INST_RETIRED:ANY
adl_grt::INST_RETIRED:ANY
```

Where adl_glc is the Alder Lake GoldenCove P-Core and adl_grt is Alder Lake Gracemont E-Core (Raptor Lake systems have the same underlying PMU as Alder Lake).

Ideally the PAPI perf_event component would already allow adding any number of perf_event supported events to the same EventSet. Unfortunately the way it is currently written EventSets can only handle events belonging to the same perf_event PMU type. You cannot have P- and E- core events in the same EventSet, nor can you have things like CPU and RAPL power events in the same EventSet. A potential workaround would be to just create two EventSets, one for the big PMU and one for the little. However this will not work as PAPI only allows one EventSet to be active per component at a time.

The solution to this issue is to modify the perf_event component so it can handle having multiple perf_event "event groups" active in one PAPI EventSet. The perf_event interface has the idea of these "event groups" that are a list of events that can be started and stopped together. However these groups also cannot contain events from different PMUs. The solution is to modify the PAPI perf_event component to track the PMU types of all events added to an EventSet, and split them up into separate perf_event groups by PMU type. Then when an EventSet is configured, started, stopped, etc., the code will operate on the multiple perf_event groups belonging to the EventSet.

The primary challenge when implementing this was finding in the code where events are manipulated and modifying these calls to search arrays of event groups when needed, adding an extra layer of indirection. The code currently uses statically allocated arrays to hold the group/PMU-type info. We will look into whether more complex data structures might provide better performance.

We have preliminary code implementing the new multi-PMU EventSet behavior. It is undergoing testing with the goal of getting it merged into the main development tree. There may be complications regarding multiplexing. On perf_event multiplexing is handled by having each event be its own event group leader while gathering some extra info. Care needs to be taken that the enhanced event support does not break existing multiplexing support.

F. PAPI support: Results

With the patches applied we have tested on both the Raptor Lake and Orange Pi systems described earlier in this paper.

We have a test: `papi_hybrid_100m_one_eventset` that runs 1 million instructions 100 times and measures the average retired events. The result should be roughly 1 million (with some minor overhead inherent in using PAPI). On a traditional machine you get the expected result.

On a heterogeneous machine with original PAPI you could specify only one of the events, so you might get 0, 1 million, or something in between depending how the OS scheduled the process. To verify we are only running on the expected core types we use the `taskset` tool to bind the process to a specific CPU core.

With the new, patched, PAPI the test runs as expected. An example result gives results like:

```
Average instructions p: 836848 e: 167487
```

showing that across the full run some instructions were on the P core, some on the E core, but if you add them up they average near 1 million as expected.

V. Future Work

While we have done the preliminary work needed to get heterogeneous CPU support into PAPI, work continues on getting all of the changes merged back into the various upstream projects. In addition there are various other tasks that we plan on contributing.

1) Detailed Processor Reporting: Currently while PAPI can detect and report the number of CPU cores and threads in a system, it currently cannot report processor type of each. We plan to add support so heterogeneous processors can be properly reported.

2) Derived Event Support: PAPI has a notion of preset events: pre-defined common event types that can be used to generically select events on any processor without having to know the name of the low-level native event. This includes things such as `PAPI_TOT_INS` which will select the proper "total retired instructions" event.

This falls apart on heterogeneous systems: which event should be chosen, the big or little one? Luckily PAPI presets can be defined as "derived" events made up of multiple underlying events. In this way a user does not have to care if they are on a heterogeneous machine. Ideally on such a machine, `PAPI_TOT_INS` and similar can be a derived event that just adds together the events for both types of cores, and transparently adds the result together behind the scenes. For example, on a Raptor Lake machine `PAPI_TOT_INS` might be generated by adding together the two events `adl_glc::INST_RETIRED:ANY` and `adl_grt::INST_RETIRED:ANY`.

This seems like it should be straightforward, but there are a few complications. On Intel processors there is only one family/model type for the overall processor, so the current way of defining presets by family/model will not work. The code that parses the `PAPI_events.csv` file will have to be modified to be aware of the existence of E and P core availability so it can properly pick which combination of events to use.

3) Remove perf_event_uncore Component: To access uncore events on Linux perf_event, PAPI has its own uncore component separate from the standard one. This is because prior to the work in this paper it was not possible to add multiple events from different perf PMUs to the same eventset. With the new infrastructure in place, it should be possible to add uncore events into combined eventset like any other event and the separate component is no longer necessary.

The question that remains is can it be removed without breaking backwards compatibility if users have uncore component names hardcoded into their workflow.

4) Unit Tests: To ensure that the new support works we need a comprehensive set of unit tests that test the new functionality. It is complicated because ideally we will cover all the tests the current does, but on all combinations of P and E-cores. This increases the surface area and will be a lot of work.

5) Overhead: The new method for adding events does an extra layer of indirection to support multiple perf event groups. The file descriptor for each group's parent has to be looked up and opened. This potentially will add overhead to the measurement process which already can be slow in some situations [25].

We need to run extensive tests to see if there are any overhead regressions. In addition, this might affect the "fast" rdpmc counter support and this needs to be tested as well.

VI. Conclusion

Heterogeneous processors are becoming increasingly common and it is only a matter of time before they arrive to the world of high performance computing. We motivate with two examples the kinds of performance issues can happen on heterogeneous systems, which shows the need for robust performance tools that can be used to analyze such problems. We are working to extend the widely-used PAPI performance library to take advantage of this knowledge and to be ready for the future world of hybrid processors.

VII. Acknowledgements

This work was supported by the National Science Foundation under Grant No. CSSI-2311709. The authors would like to thank Connor Noddin for preliminary work done characterizing the Raptor Lake processor. We would also like to thank Stephane Eranian for all his help in getting hybrid event support for both x86 and ARM added to the libpfm4 library used by PAPI.

References

[1] A. Phillips, "ARM unveils its most energy efficient application processor ever; redefines traditional power and performance relationship with big.LITTLE processing," https://web.archive.org/web/20111221230916/http://www.arm.com/about/newsroom/arm-unveils-its-most-energy-efficient-application-processor-ever-with-biglittle-processing.php, Oct. 2011.

[2] "big.LITTLE technology: The future of mobile," ARM, 2013.

[3] E. Rotem, A. Yoaz, L. Rappoport, S. Robinson, J. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel Alder Lake CPU architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, Apr. 2022.

[4] M. Larabel, "AMD posts patches for improving heterogeneous core type CPUs on Linux," https://www.phoronix.com/news/AMD-Heterogeneous-P-State-Linux, May 2024.

[5] A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs, "Scheduling heterogeneous processors isn't as easy as you think," in *In Proc. 2012 ACM-SIAM Symposium on Discrete Algorithms*, 2012, pp. 1242–1253.

[6] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. of 39th Annual International Symposium on Computer Architecture*, Jun. 2012.

[7] C. V. Li, V. Petrucci, and D. Mossé, "Predicting thread profiles across core types via machine learning on heterogeneous multiprocessors," in *In Proc. VI Brazilian Symposium on Computing Systems Engineering*, Nov. 2016.

[8] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal, "A machine learning approach for performance prediction and scheduling on heterogeneous CPUs," in *In Proc. 29th International Symposium on Computer Architecture and High Performance Computing*, Oct. 2017.

[9] M. Sagi, M. Rapp, H. Khdr, Y. Zhang, N. Fasfous, N. A. Vu Doan, T. Wild, J. Henkel, and A. Herkersdorf, "Long short-term memory neural network-based power forecasting of multi-core processors," in *In Proc. Design, Automation & Test in Europe Conference & Exhibition*, 2021.

[10] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Department of Defense HPCMP User Group Conference*, Jun. 1999.

[11] T. Gleixner and I. Molnar, "Performance counters for Linux," 2009.

[12] A. Petitet, R. Whaley, J. Dongarra, A. Cleary, and P. Luszczek, "HPL — a portable implementation of the high-performance linpack benchmark for distributed-memory computers," Innovative Computing Laboratory, Computer Science Department, University of Tennessee, v2.2, http://www.netlib.org/benchmark/hpl/, Dec. 2017.

[13] "OpenBLAS an optimized BLAS library website," https://www.openblas.net/.

[14] "Developer guide for Intel® oneAPI math kernel library linux," https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-linux/2024-2/overview.html, Jun. 2024, [Accessed: July 22, 2024].

[15] Z. Krpić, L. Loina, and T. Galba, "Evaluating performance of SBC clusters for HPC workloads," in *In Proc. International Conference on Smart Systems and Technologies*, Oct. 2022, pp. 173–178.

[16] E. Rotem, A. Naveh, D. Rajwan, A. Anathakrishnan, and E. Weissmann, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, 2012.

[17] "Intel® core™ i7-13700 processor specifications," https://www.intel.com/content/www/us/en/products/sku/230490/intel-core-i713700-processor-30m-cache-up-to-5-20-ghz/specifications.html, [Accessed: July 25, 2024].

[18] S. Stepanovic, G. Georgakarakos, S. Holmbacka, and J. Lilius, "Quantifying the interaction between structural properties of software and hardware in the ARM Big.LITTLE architecture," in *In Proc. 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Mar. 2018.

[19] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, "DyPO: Dynamic pareto-optimal configuration selection for heterogeneous MpSoCs," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 123, pp. 1–20, Sep. 2017.

[20] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. Katevenis, "Modeling energy-performance tradeoffs in ARM big.LITTLE architectures," in *In Proc. 27th International Symposium on Power and Timing Modeling, Optimization and Simulation*, Sep. 2017.

[21] J. Whitehouse, Q. Wu, S. Song, E. John, A. Gerstlauer, and L. K. John, "A study of core utilization and residency in heterogeneous smart phone architectures," in *In Proc. ACM/SPEC International Conference on Performance Engineering*, Apr. 2019, pp. 67–78.

[22] G. Fernandez, F. J. Cazorla, J. Abella, and S. Girbal, "Assessing time predictability features of ARM Big.LITTLE multicores," in *In Proc. 30th International Symposium on Computer Architecture and High Performance Computing*, Sep. 2018.

[23] R. Neri, "[patch 0/4] drivers core: Introduce CPU type sysfs interface," https://lkml.org/lkml/2020/10/2/1208, Oct. 2020.

[24] S. Eranian, "libpfm4: improving performance monitoring on Linux," http://perfmon2.sourceforge.net/.

[25] V. Weaver, "Self-monitoring overhead of the linux perf_event performance counter interface," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2015.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 Code to enable heterogeneous support for PAPI.
- C_2 A comparison of HPL compiled with OpenBLAS and Intel's Optimized HPL on a heterogeneous CPU.
- C_3 A comparison of HPL running on an OrangePi big.LITTLE system.

B. Computational Artifacts

A_1 https://zenodo.org/doi/10.5281/zenodo.13287670

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3	Tables 1-2 Figures 1-4
..		

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact is the code needed to patch libpfm4 and PAPI to enable heterogeneous support.

Expected Results

Expected results are a version of PAPI with heterogeneous support.

Expected Reproduction Time (in Minutes)

Should take less than an hour.

Artifact Setup (incl. Inputs)

Hardware: Should work on any Linux system, but ideally test it on a heterogeneous system such as Raptor Lake or an ARM big.LITTLE system.

Software: Requires standard Linux distribution and git checkouts of the libpfm4 and PAPI development trees.

Datasets / Inputs: n/a

Installation and Deployment: The included README in the patches directory describes the versions of the packages needed and how to apply the patches.

Artifact Execution

Requires gcc compiler.

Artifact Analysis (incl. Outputs)

Output should be working version of PAPI. The test provided in the tests directory will show if heterogeneous support is working on a Raptor Lake or OrangePi 800 big.LITTLE machine.

B. Computational Artifact A_2

Relation To Contributions

The artifact (the mon_hpl.py script and its supporting scripts) was used to acquire data to compare HPL compiled with OpenBLAS to Intel's Optimized HPL.

Expected Results

The data collected by the artifact when running Intel HPL should indicate better performance than when running OpenBLAS HPL. The data collected by the artifact enables the comparison of the two benchmarks in a number of different metrics including CPU frequency, thermal performance, energy consumption, and performance counter values.

Expected Reproduction Time (in Minutes)

The expected computation time for this artifact is given by Equation 1, where t_{HPL} is the expected amount of time for a single run of HPL on the system, $t_{thermal}$ is the expected amount of time it will take for the thermal zones to settle, and N is the number of runs.

$$t_{A2} = (t_{HPL} + t_{thermal}) * N \quad (1)$$

Artifact Setup (incl. Inputs)

Hardware: For RAPL data to be collected, an Intel CPU supporting RAPL is required.

Software: The scripts were designed for Python 3.11.2 and require the Python packages "numpy" version 1.24.2 and "matplotlib" version 3.6.3. The scripts are designed to run in a linux environment, and in order to collect RAPL data root permissions are required. Additionally, a version of HPL must be installed for the script to run HPL.

Installation and Deployment: To install and deploy, follow the instructions in the provided Readme.md file of the artifact.

Artifact Execution

The workflow of the artifact is described by two tasks $T_1 \rightarrow T_2$. In T_1, the mon_hpl.py script is ran with the desired parameters to generate a directory of raw data. In T_2, the process_runs.py script is ran with the raw data as an input, and produces processed data that can subsequently be plotted or analyzed.

The experimental parameters usable by mon_hpl.py are passed as command line arguments, and their values necessary to reproduce the artifact on the experiment's hardware are listed in Table I.

TABLE I
PARAMETERS USED BY MON_HPL.PY

Parameter	Value
–n_runs	10
–cores	0,2,4,6,8,10,12,14,16-24
–settled_temps	thermal_zone9:35000

“thermal_zone9” corresponds to the thermal zone type “x86_pkg_temp”. All other parameters were left at their default values.

Artifact Analysis (incl. Outputs)

The output of the artifact is recorded CPU frequency, thermal zone values, RAPL energy data, and performance counter data. The output is described in further detail in the artifact’s Readme.md file.

C. Computational Artifact A_3

Relation To Contributions

This artifact is the results from running HPL on the OrangePi system.

Expected Results

Expected results are similar HPL results assuming run on an identical OrangePi system.

Expected Reproduction Time (in Minutes)

The actual runs to replicate might take a while (multiple hours) as time is taken for the CPU to cool off after each run.

Artifact Setup (incl. Inputs)

Hardware: Requires an OrangePi 800 system. For the power measurements you will need a WattsUpPro power meter.

Software: Requires Ubuntu distribution that comes with the OrangePi 800.

Datasets / Inputs: The HPL binary used is provided. Also included is the HPL.dat input file.

Installation and Deployment:

Artifact Execution

n/a

Artifact Analysis (incl. Outputs)

Experiments involved running the provided HPL binary on the system and configuring it to limit the cores involved with `env OMP_NUM_THREADS` and the thread to bind to with `taskset`.

The raw data results are provided, as well as README files with more info as well as info on how the plots were made.